View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Scholarly Materials And Research @ Georgia Tech

SECURE AND HIGH-PERFORMANCE BIG-DATA SYSTEMS IN THE CLOUD

A Thesis
Presented to
The Academic Faculty

by

Yuzhe Tang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science, College of Computing

Georgia Institute of Technology
August 2014

Copyright © 2014 by Yuzhe Tang

SECURE AND HIGH-PERFORMANCE BIG-DATA SYSTEMS IN THE CLOUD

Approved by:

Professor Ling Liu, Committee Chair
School of Computer Science, College of Computing
Georgia Institute of Technology

Professor Ling Liu, Advisor
School of Computer Science, College of Computing
Georgia Institute of Technology

Professor Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Professor Doug Blough
School of Electronic and Computer Engineering
Georgia Institute of Technology

Professor Edward R. Omiecinski
School of Computer Science
Georgia Institute of Technology

Date Approved: May 2014

Dedicated to Xi

ACKNOWLEDGEMENTS

First of all, I want to thank my advisor, Prof. Ling Liu, for her guidance and support throughput my PhD studies. Prof. Liu introduced me to my thesis area, and gave me the freedom to explore various topics. Her vision and insights have proven to be invaluable. Besides, Prof. Liu is very supportive and patient, which is very helpful when I encounter obstacles. I will always cherish the opportunity working with her. I would like to express my deepest gratitude to Prof. Mustaque Ahamad, Prof. Doug Blough, Prof. Calton Pu, and Prof. Edward Omiescinski for serving on my thesis committee. Their insightful feedback has significantly improved the thesis research. Their guidance always pushes me to think deeper and sharper and their enthusiasm on carrying out research activities greatly influences me. I am also very grateful to Prof. Sham Navathe for serving in my qualifier committee; Prof. Navathe gave me a number of suggestions on both learning and research. Dr. Ada Gavrilovska helped me and gave me guidance on conducting system-oriented researches, and I deeply appreciate all her help.

Dr. Bugra Gedik was my mentor during my first summer internship in IBM Research at Hawthorne, NY. Dr. Gedik is one of the most hard-working people I have ever met. His philosophy for picking research problems has a great influence on me, and I really appreciate this wonderful experience working with him. Dr. Arun Iyengar and Dr. Wei Tan were my mentors during my second internship in IBM Research, Yorktown Heights, NY. Dr. Iyengar held many meetings with me for research-oriented discussions and was very patient and responsive whenever I needed his help. Dr. Wei Tan worked with me on a daily basis and his profound knowledge on cloud systems has impacted my research since then. I enjoyed the conversations with Dr. Tan in the working places and personal circumstances. Dr. Ting Wang and Dr. Xin Hu were my mentors during my third internship in IBM

Research, Yorktown Heights, NY. Dr. Ting Wang was very encouraging on me pursuing my research problems and I am grateful for having the flexibility. Dr. Xin Hu gave me tremendous support for for shaping the research topic and I enjoyed the conversations with him a lot.

During my PhD studies in Atlanta, I had the privilege to both work and live with many brilliant people. I sincerely thank my fellow students in the DiSL research group and in the College of Computing, including Bhuvan Bamba, Ting Wang, Peter Pesti, Gong Zhang, Qinyi Wu, Sankaran Sivathanu, Myungcheol Doo, Shicong Meng, Fang Zheng, Qingyang Wang, Balaji Palanisamy, Binh Han, Yiduo Mei, Rui Zhang, Liting Hu, Kisung Lee, Qi Zhang, Emre Yigitoglu and others. They helped and supported me in many ways, and made my stay at Atlanta a lot more fun. I will always cherish this journey with them.

Last but not least, I would like to thank my girl friend and my parents for their love and support. My girl friend, Xi Liu, is the source of my inspiration and my passionate for life; You are the person I love and I can never be more grateful for your presence in my Ph.D. life. My parents, Yunxia Tang and Hui Peng, always give me unconditional love in their own way. Their love and guidance shape me into the person I am. This thesis is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Cloud computing and big data technology continue to revolutionize how computing and data analysis are delivered today and in the future. To store and process the fast-changing big data, various scalable systems (e.g. key-value stores and MapReduce) have recently emerged in industry. However, there is a huge gap between what these open-source software systems can offer and what the real-world applications demand. First, scalable key-value stores are designed for simple data access methods, which limit their use in advanced database applications. Second, existing systems in the cloud need automatic performance optimization for better resource management with minimized operational overhead. Third, the demand continues to grow for privacy-preserving search and information sharing between autonomous data providers, as exemplified by the Healthcare information networks.

My Ph.D. research aims at bridging these gaps.

First, I proposed HINDEX, for secondary index support on top of write-optimized key-value stores (e.g. HBase [14] and Cassandra [9]). To update the index structure efficiently in the face of an intensive write stream, HINDEX synchronously executes append-only operations and defers the so-called index-repair operations which are expensive. The core contribution of HINDEX is a scheduling framework for deferred and lightweight execution of index repairs. HINDEX has been implemented and is currently being transferred to an IBM big data product.

Second, I proposed Auto-pipelining for automatic performance optimization of streaming applications on multi-core machines. The goal is to prevent the bottleneck scenario in which the streaming system is blocked by a single core while all other cores are idling, which wastes resources. To partition the streaming workload evenly to all the cores and

to search for the best partitioning among many possibilities, I proposed a heuristic based search strategy that achieves locally optimal partitioning with lightweight search overhead. The key idea is to use a white-box approach to search for the theoretically best partitioning and then use a black-box approach to verify the effectiveness of such partitioning. The proposed technique, called Auto-pipelining, is implemented on IBM Stream S.

Third, I proposed ϵ-PPI, a suite of privacy preserving index algorithms that allow data sharing among unknown parties and yet maintaining a desired level of data privacy. To differentiate privacy concerns of different persons, I proposed a personalized privacy definition and substantiated this new privacy requirement by the injection of false positives in the published ϵ-PPI data. To construct the ϵ-PPI securely and efficiently, I proposed to optimize the performance of multi-party computations which are otherwise expensive; the key idea is to use addition-homomorphic secret sharing mechanism which is inexpensive and to carry out the distributed computation in a scalable P2P overlay.

CHAPTER I

INTRODUCTION

1.1 Background Introduction

In the last decade, we witnessed the advent and rise of the era of cloud computing and big data. Cloud computing has dramatically changed the way we live our lives. Many human activities are nowadays supported online and in the cloud, ranging from social networking, shopping, Healthcare, and even political campaigns. The new needs for big-data computation in the cloud has given rise to a new market for cloud services: On the front end, many web companies emerge to support domain-specific human activities, such as Foursquare for social networking, Amazon.com for shopping and so on; On the back end, various generic cloud service providers come into being to serve the cloud workload for the front-end companies in a cost-effective way. The most famous cloud service providers include Amazon AWS, Microsoft Azure, and IBM SmartCloud.

A typical cloud system, illustrated in Figure 1, interacts with three external parties: a cloud service provider for provisioning the computing resources on which the cloud system is up and running, a small front-end web company that administrates and manages the cloud system, and a large number of end users who use the end service provided by the cloud system and delegate their personal data to the cloud. A real-world example following this new computing paradigm is the Foursquare case, in which the front-end webpage (i.e. Foursquare.com) serves mobile social users using the cloud resources provisioned by Amazon AWS.

A cloud cluster usually spans across multiple domains. For example, in Amazon AWS, there are multiple administrative domains, called availability zones, each dedicated for serving users of a geographic region (e.g. North America, Eastern Asia, etc). Inside each

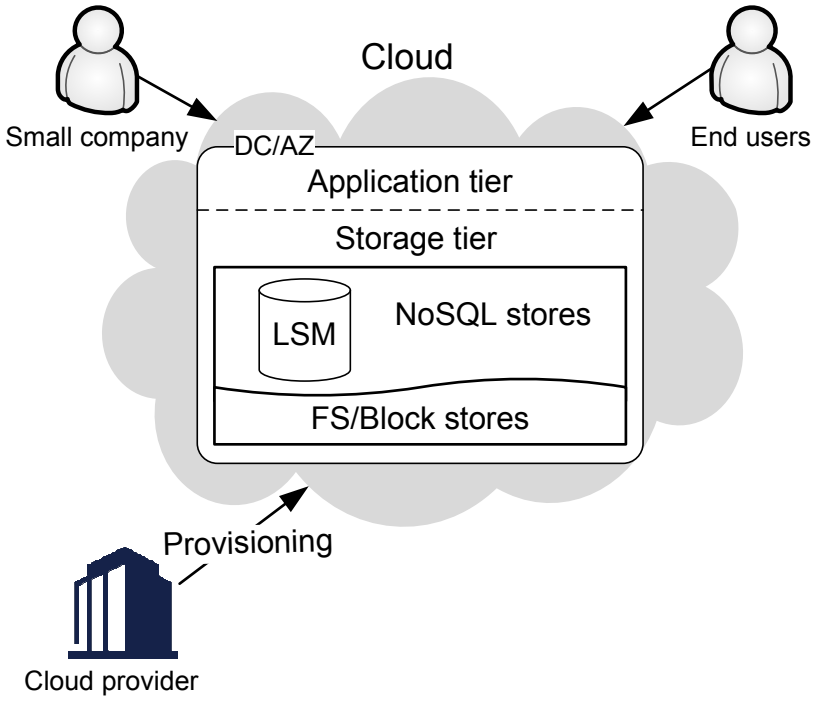


Figure 1: A typical system in a cloud

availability zone, there are multiple data centers geo-distributed at different locations. A domain can also be a service provider itself. While a small company may delegate its own data to one or more cloud services, cloud services are under different and separate administrative domains which present security challenge for data sharing between domains.

A cloud system internally consists of a large computer cluster which is organized into multiple tiers, typically, with a web tier, an application tier, and a storage tier. To support functionality at different tiers, various scalable software systems emerge in the age of cloud computing. Most of them fall under two categories: 1) big data processing systems such as MapReduce [61]/ Hadoop, Dryad [81] and Hive [8, 124], and 2) big data storage systems, such as the increasingly popular NoSQL stores (also called key-value stores [53, 14, 9]) and NewSQL systems [57, 43]. My thesis research primarily focuses on the big data storage systems.

Many key-value stores deal with write-intensive data. In many modern cloud applications, data is innately write-intensive. In the Web 2.0, for example, average users not only

read the news, but also contribute their own ideas and write news by themselves in the form of social status, blogs, etc. Write-intensive data is also widely observed in the application areas of financial trading, online gaming and etc. Therefore, it is desirable to optimize the write performance for cloud software systems, particularly in the case of key-value stores. Many key-value stores (including BigTable [53], HBase [14], Cassandra [9], LevelDB [10], etc) adopt a log-structured merge tree or LSM [99] for optimized random-write performance. By the design principle, LSM is a system that makes a trade-off between purely log-structured systems (e.g. LFS [105]) and update-in-place database systems (e.g. MySQL).

Under this perspective for cloud computing, I address many new challenges for designing secure and high-performance cloud systems, and propose several new technical solutions.

1.2 Challenges

1.2.1 Trust and Security

The cloud computing essentially requires the full user trust and this raises controversies. On the one hand, the cloud service providers always claim themselves being trustworthy; "Do not be evil" is a common slogan used by Google and many other cloud providers. On the other hand, a large number of data-loss incidents happen every day in the cloud; for instance, user payment information in the Target store was massively disclosed in Dec. 2013 [11]. A cloud service provider may find it difficult to maintain trustworthiness due to multiple factors: 1) Under the pressure of U.S. government, a cloud service provider may have to turn in the personal user information for national security purposes at the expense of user privacy. Consider the recent PRISM scandal where NSA (i.e. National Security Administration) collects personal information through various channels of networks and big-data companies without notifying the data owners. 2) The systems and networks in the cloud are error prone; what is particularly common in large-scaled data centers is system

failure, power outage and etc. To stay trustworthy, the cloud service providers need to be fault tolerant, which is challenging: In the real life, cloud users may experience the constant unavailability of their data in the cloud. Recently in Jan. 2014, GMail has been down more than one hour during which period of time GMail users in the global can not access their personal email online. 3) Cloud service providers are driven by profit and have the incentive to be "evil". For example, Facebook has once changed its term of services, trying to own the copyright of its users' social presence data (e.g. status updates and blogs). In that case, the cloud users' data do not belong to the users any more. 4) Given a large number of hackers on the Internet, the design of cloud systems has to be secure and resilient to various attacks, which by itself presents a huge challenge.

While we have seen that it is unlikely, if not total impossible, to assume a trustworthy cloud, there is a new research area based on the untrusted cloud. It considers that an average cloud user should submit his/her data to the cloud with data fully encrypted. To enable the regular query processing in the cloud, the emerging research focuses on the technique for "querying encrypted data". Representative work includes CryptDB [102] from the system community and FHE [72] from the cryptography community; in particular, FHE (fully homomorphic encryption) is a recent theory breakthrough which enables generic computations over encrypted data. However, along this line of research, a big problem is the performance overhead. The computation cost of current FHE techniques is so high that they prohibit practical use.

Given the drawbacks of assuming a cloud is either entirely trustworthy or entirely not, a more practical model for the cloud system is a hybrid trust model. Given multiple domains in the cloud, some service providers or domains are trusted while others are not trusted. A cloud user has the freedom to choose which service providers to trust. A representative system under this trust model is the information networks. In an information network, a cloud user can choose to trust a service provider and store his/her personal data there. The

providers have the responsibility to protect user privacy as regulated by various domain-specific laws (e.g. HiPAA [7] for protecting Healthcare data privacy). A real-world example of information networks is the emerging Health Information Exchange systems (e.g. NHIN [24], GaHIN [6] and CommonWell [3]), in which each hospital is considered as a service provider which stores patients' medical records on their behalf. From an application point of view, information exchanges between different autonomous providers are critical. For example, without information exchange, a doctor may not be able to quickly find the patient's history of hospital visits for immediate and accurate medical diagnosis and treatment. For another example, a research facility may want to learn the pattern of disease outbreak which involves aggregating patient information across multiple hospitals. However, privacy issues are raised when information is exchanged across autonomous domains, which may disclose private information to unintended hospitals. It is therefore desirable to preserve user privacy and information confidentiality during the information exchanges.

1.2.2 Performance

In the age of cloud computing, scalable software systems run data-intensive workloads on a large number of commodity machines. The design of these systems achieves scalability at the expenses of simplified functionality. For example, existing key-value stores support only Put/Get alike API, and considers a simple key-value data model. While this suffices to support the basic web applications, it may fall short when the applications become complicated, such as database applications which require complex SQL support. It thus calls for scalable SQL support in the cloud.

Today's cloud workload is typically data intensive and dominated by data writes mostly. To be able to persist the intensive data writes efficiently, append-only design is commonly applied in the cloud system, which aims at transferring random disk accesses to sequential ones in order to save disk seeks. However, the append-only design usually sorts data based on arrival time and thus destroys the data locality; data versions of the same key are spread

across multiple places on disk, which in turn slows down the data-read performance. This is especially the case for supporting SQL queries that involve various data access methods. Therefore, it presents a challenge to optimize the query performance in terms of both reads and writes in the data-intensive applications.

In a cloud system of massive number of machines, operational cost could be prohibitively high due to the constant system failure and software misconfiguration which require human involvement. To improve the operational efficiency, automation is crucial. In particular, automatic performance optimization is a challenging task. On the one hand, there are so many possibilities in configuring the distributed large software in the cloud. To search for the best configuration for a particular workload, it entails iterating through all possibilities which can be a heavy and daunting task. On the other hand, workloads in a cloud are dynamic and changing in nature, which requires the performance optimization component to quickly find the best configuration and to catch up with the workload changes in real time. In other words, the performance optimization should be lightweight and responsive. Therefore, it presents a challenge to design a lightweight online optimization scheme for large-scale cloud systems supporting dynamic workloads.

1.3 Thesis Statement

Motivated by the importance and challenges of secure and high-performance big-data services in the cloud, the goal of this dissertation is to demonstrate the following thesis statement:

The big-data storage and serving in the cloud can be made secure and high-performance in the presence of multi-domain cloud infrastructure, write-intensive database workloads and large scale distributed systems.

1.4 Technical Contributions

1.4.1 Extending Key-Value Stores with Efficient Secondary-Index Support

I proposed HINDEX to extend the key-value stores' API to support value-based data access. Specifically, the Put/Get API exposed by existing key-value stores allows only key-based data access in the sense that data key k is a required parameter as in the API calls, Put(k, v) and Get(k)$\rightarrow v$. Given that many database and web applications require value-based access (e.g. in an SQL query, an attribute value can be used in the WHERE clause), HINDEX aims at adding value-based access methods to the existing key-value stores with a proposed new API: ReadValue(v)$\rightarrow \{k\}$. Given that data in key-value stores is of multiple overwriting versions [62], ReadValue is designed to return the latest and freshest version in real time.

In the system design of HINDEX, the challenges come from the performance aspect. In order to guarantee that ReadValue returns the latest version and the index structure is always fresh, it is necessary to read and delete the old data versions upon a data update. Given the log-structured design of key-value stores, the performance of a write (or a Put) is optimized at the expenses of the read (or Get) performance. Therefore the conventional way to update index structure which requires reads would greatly slow down the data ingest rate. The technical contribution of HINDEX is a scheduling framework that executes the expensive read-and-delete operations in a lightweight fashion. I proposed an offline scheduling strategy which co-schedules the read-and-delete operations with a compaction, which is a native offline process in many key-value stores that reorganizes the on-disk data layout and cleans up data garbage. Based on this new scheduling strategy, the write performance of indexed system is significantly improved, as verified by our real-world experiment evaluation.

HINDEX has been implemented on HBase. Based on collaborative work with an IBM colleague, part of the HINDEX technique is currently being transferred [113] to an IBM big-data product, BigInsights [1].

1.4.2 Searching Information Networks with Personalized Privacy Preservation

In information networks, data sharing is established by first searching for a provider of interest. Such search may disclose private information, since the search result contains the sensitive linkage information regarding which provider a person has delegated his/her data to. In the health domain, for example, such information discloses a patient's history of hospital visits which is deemed sensitive and private. While existing search facilities in the market, most notably record locator service [25, 4], do not address privacy concerns, I proposed a privacy-preserving index, called ϵ-PPI [116], for privacy-aware search in the information networks. In particular, ϵ-PPI personalizes privacy preservation to address the different privacy concerns regarding different data owners (e.g. patients). Towards this goal, I proposed a noise-based technique to control the confidence that an attacker can have when attacking based on the ϵ-PPI search result.

Under this framework, I discovered a generic vulnerability and devised a new attack named the common-identity attack that breaks existing privacy-reserving index systems. A common identity is one that belongs to a person who has the data stored on almost all providers in the network and its existence could potentially leak private information. I proposed a technique based on the idea of mixing identities of different kinds in order to provide quantitative privacy guarantees for the common identities.

In addition, I proposed a performance-optimized protocol to securely construct ϵ-PPI in a mutually-untrusted network. For secure distributed computations, the common wisdom is to use MPC or secure Multi-Party Computations [130, 95, 47]. MPC is very expensive for bit-wise computations and directly applying MPC to our ϵ-PPI construction problem would incur prohibitively high cost as an information network is typically large scaled in the sense of both data size (e.g. a hospital may generates 1 TB data daily) and network scale (e.g. in the U.S., there are around six thousand hospitals). The contribution that I made in constructing ϵ-PPI is a performance-optimized protocol that requires no trusted party. ϵ-PPI applies secret sharing schemes to reduce the amount of computation carried

out by MPC; the insight is that secret sharing schemes, which protecting data secrecy, are homomorphic to addition and much more efficient than the cryptographic MPC.

1.4.3 Optimizing Streaming Performance on Multi-core using Automatic Pipelining

In a cloud system, application servers would use multi-core machines to serve streaming applications. A typical streaming application is modeled as an operator graph in which each vertex represents a set of computations and each edge represents a data flow. Running streaming applications with multiple cores requires the operator graph to be partitioned to pieces of workloads, each piece scheduled on one core. To optimize the performance, the goal is to prevent a so-called single-core bottleneck situation in which the whole system is blocked by a single busy core while other cores are idling, which thus wastes resources. In this case, the problem of streaming system optimization boils down to searching for the best partitioning scheme that could evenly assign the streaming workload to the cores. Given a large operator graph, there are a large number of possible ways in partitioning the graph, which presents a huge search space for the best partitioning. Therefore, it is challenging to find the optimal partitioning with low search overhead.

In this context, my contribution is an efficient search strategy for locally optimal partitioning scheme with verified performance improvement. In a nutshell, my search strategy is a combination of "white-box" and "black-box" approaches. The white-box approach models the streaming application as an execution graph and searches for the optimal solution efficiently based on a greedy strategy and heuristic-based utility. Given the search result, it then tries the "black-box" approach; it verifies the performance gain when the system is reconfigured based on the search result of the white-box approach. If it does not improve the performance, then it would withdraw and try the second-optimal search result. Otherwise, the new partitioning strategy stays.

The proposed technique, called Auto-pipelining [114], is implemented on IBM Stream S, including a component for lightweight CPU utilization monitoring, a decision making

framework based on the search algorithm, and an action taking component which substantiates the partitioning strategy using Stream S's native API.

1.5 Organization of the Dissertation

The reminder of this thesis dissertation is organized as follows.

Chapter II presents the ϵ-PPI framework for privacy-preserving search and indexing over multiple data providers in the cloud.

Chapter III presents the HINDEX system for supporting big-data indexing on scalable key-value stores in the cloud.

Chapter IV presents Auto-Pipelining which automatically optimizes the performance of streaming applications on multi-core machines.

Chapter V presents a summary of my thesis research and discusses several open problems for future researches.

CHAPTER II

ϵ-PPI: PRIVACY-PRESERVING SEARCH IN MULTI-DOMAIN INFORMATION NETWORKS

An information network is a model for cloud computing, in which multiple service providers are under different administrative domains and are mutually untrusted. In this chapter, I describe the ϵ-PPI work which addresses the privacy issue when data needs to be shared across the domain boundary in information networks in the cloud.

2.1 Introduction

In information networks, autonomous service providers store private personal records on behalf of individual owners and enable information sharing under strict enforcement of access control rules. Such information networks have the following salient features: 1) Providers, each under a different administrative domain, do not mutually trust each other; 2) Providers have the responsibility of protecting owners' privacy.

An example of the information network is the emerging HIE or Healthcare Information Exchange systems (e.g. NHIN [24], GaHIN [6] and CommonWell [3]), in which patients delegate their personal medical records to the hospitals that they visited and hospitals form a nation-wide (or state-wide) network to share information. Specifically, different hospitals may compete for the same customer base (i.e. patients) and have conflicting economic interests, which renders it difficult to build full trust relationships between them. Hospitals are responsible for protecting patient privacy, as regulated by Federal laws (e.g. HiPAA [7]). Other examples of the multi-domain information networks include cross-university online course management systems (e.g. Coursera [5] and StudIP [31]), distributed social networks (e.g. Diaspora [59] and Twister [32]) and others.

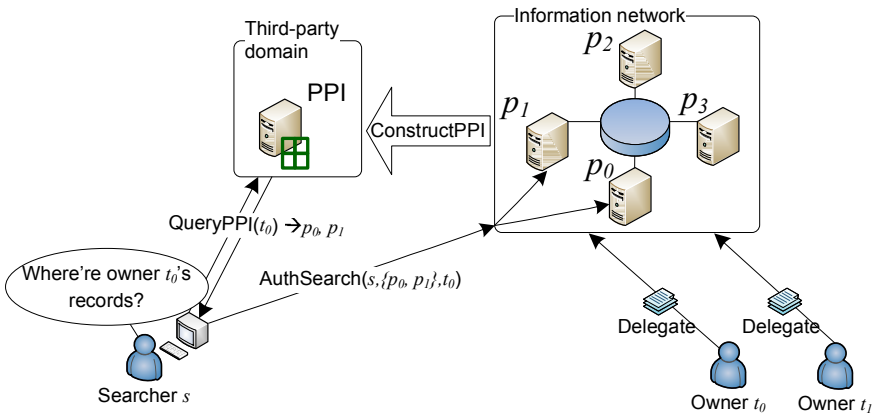


Figure 2: The system of PPI and information network

Information sharing is crucial for various applications in information networks. In the HIE case, for example, when a patient who is unconscious is sent to a hospital, information sharing between multiple hospitals can help the doctor retrieve the patient's medical history for immediate and accurate medical treatment. To establish information-sharing sessions, Record Locator Service [4, 25] is a standard procedure in the existing HIE systems; it provides the ability to identify where a patient's records are located based upon her identity, and is used as the first step towards sharing information between the searcher and the patient's hospital of interest. Internally, a locator service maintains meta-data regarding the patients' medical history (i.e. the membership between a patient and a hospital). Such meta-data is private and sensitive by itself; for example, the fact that a sports celebrity visited a hospital before is something that s/he wants to keep confidential since disclosing it may jeopardize his/her future career. A locator service is usually hosted by an untrusted third-party entity, mainly because of the difficulty to find a party unanimously trusted by all the autonomous providers [1]; for example, consider the U.S. government as a candidate, but various scandals including the recent PRISM program [29] have made the government lose the public trust. It is therefore desirable to preserve data privacy when the locator service is hosted by an untrusted entity.

PPI techniques (i.e. privacy preserving index [45, 44, 131]) is promising for preserving

[1] In this chapter, we will use "provider" and "hospital" interchangeably.

privacy of a locator service. The PPI design, while originally proposed in domains other than information networks, could protect patient's privacy regarding the medical-history meta-data. When applying PPI for the locator service, the working (also elaborated in Section 2.2) is a two-phase search procedure. Illustrated in Figure 2, a searcher for records of a certain owner [2] first queries the PPI, and obtains a list of providers that may or *may not* have the records of interest. Then for each provider in the list, the searcher attempts to get authenticated and authorized before she can locally search the private records there. In a PPI, the privacy preservation comes from the fact that a searcher may encounter in a PPI result some noise providers (from which she does not find any matching data).

2.1.1 Quantitatively Personalized Privacy Preservation

While existing PPI's have addressed privacy preservation, none of these approaches recognize the needs of personalized privacy, that is, to personalize privacy preservation for different owners and providers. Recall that the privacy of a PPI system is about "an owner t_j has the records stored on provider p_i". It is evident that disclosing the private fact regarding different owners and providers causes different levels of privacy concerns. For example, a woman may consider her visit to a women's health center (e.g., for an abortion) much more sensitive than her visit to a general hospital (e.g., for cough treatment). Similarly, different owners may have different levels of concerns regarding their privacy: While an average person may not care too much about his/her visit to a hospital, a celebrity may be much more concerned about it, because even a small private matter of a celebrity can be publicized by the media (e.g., by paparazzi). It is therefore critical to personalize privacy protection in a PPI system. That being said, using existing PPI approaches can not provide quantitative guarantees on the privacy preservation degree, let alone on a personalized basis. The cause, largely due to the privacy-quality-agnostic way of constructing PPI systems, is analyzed in Section 2.2.4.

[2] In this chapter we will use "owner" and "patient" interchangeably.

In this chapter, we propose a new PPI abstraction for quantitatively personalized privacy control, coined ϵ-PPI. Here, ϵ is a privacy-aware knob that allows each owner to mark a personalized privacy level for a data unit delegated to the providers. Specifically, ϵ_j is a value in a spectrum from 0 to 1, where value 0 is for the least privacy concern (in this case, the PPI returns the list of the "true positive" providers who truly have the records of interest) and value 1 for the best privacy preservation (in this case, PPI returns all providers, and a search is essentially broadcast to the whole network). By this means, an attacker observing the PPI search result can only have a bounded confidence by ϵ in successfully identifying a true positive (and vulnerable) provider from the obscured provider list.

To construct the new ϵ-PPI from a network of mutually untrusted providers, we rely on MPC technique (i.e., secure multi-party computation [95, 47, 79, 60]) which addresses the input-data privacy in a generic computation process. However, by directly applying MPC to our ϵ-PPI-construction problem, it raises performance issues. On the one hand, current MPC platforms can only scale to small workloads [97]; they are practical only for simple computation among few parties. On the other hand, a typical ϵ-PPI construction may involve millions of owners and thousands of providers (e.g. in United States there are about six thousand hospitals), which entails an intensive use of bit-wise MPC. It is therefore critical to devise a practical MPC protocol to efficiently carry out the ϵ-PPI construction. In this regards, we propose to minimize the expensive MPC by using a parallel secure sum protocol. The secure sum can be efficiently carried out by a proposed secret sharing scheme with additive homomorphism. Based on the proposed MPC primitive, our index construction protocol protects providers' privacy and can tolerate collusion of up to c providers (c is configurable).

The contributions of this chapter are following:

- We propose ϵ-PPI that personalizes the privacy protection with quantitative guarantees. The ϵ-PPI exposes a new delegate operation to owners, which allows them to specify their different levels of privacy concerns. This new privacy knob, coined ϵ,

can give quantitative privacy control while enabling information sharing.

- We propose ϵ-PPI construction protocol for an untrusted environment. As far as we know, this is the first PPI construction protocol without assumption on trusted parties or mutual trust relationships between providers. The performance of ϵ-PPI construction protocol is extensively optimized by reducing the use of costly generic MPC and using the proposed domain-specific protocols. The proposed construction protocol is implemented and evaluated with verified performance superiority.

- We introduce a new privacy attack (called common-identity attack) that can break generic PPI systems. The new attack model targets vulnerable common owners/patients who visited a large number of hospitals. Our proposed ϵ-PPI is the first to resist common-identity attacks by using a proposed term-mixing protocol.

The rest of this chapter proceeds as follows: Section 2.2 formulates the ϵ-PPI problem. Section 2.3 and 2.4 respectively describe the computation model and distributed implementation of the ϵ-PPI construction protocol. Section 2.5 presents evaluation results, and Section 2.6 surveys the related work before the conclusion in Section 4.11.

2.2 Problem Formulation

2.2.1 System Model

We formally describe our system model, which involves four entities: 1) a set of n data owners, each of whom, identified by t_j, holds a set of personal records, 2) an information network consisting of m providers in which a provider p_i is an autonomously operating entity (e.g. a hospital), 3) a global PPI server in a third-party domain, 4) a data searcher who wants to find all the records of an owner of interest. The interactions between these four entities are formulated by the following four operations.

- Delegate($\langle t_j, \epsilon_j \rangle$, p_i): A data owner t_j can delegate his/her records to provider p_i based on the trust relationship (e.g. such trust can be built by the previous visit to

a hospital). Along with the record delegation, the owner can specify her personal preference in privacy by degree ϵ_j. Here ϵ_j indicates the level of privacy concerns, ranging from 0 up to 1. For example, a VIP user (e.g. a celebrity patient in the eHealthcare network) may want to set the privacy level at a high value while an average patient may set the privacy level at a medium value [3].

- ConstructPPI($\{\epsilon_j\}$): After data records are populated, all m providers in the network join a procedure ConstructPPI to collectively construct the privacy preserving index. The index construction should comply with owner-specified privacy degree $\{\epsilon_j\}$. As will be elaborated, the constructed PPI contains noises or false positives for the purpose of privacy preservation and $\{\epsilon_j\}$ is materialized as the false positive rate of owner t_j.

- QueryPPI(t_j)$\rightarrow \{p_i\}$: At the service time, a searcher s, in the hope of finding owner t_j's records, initiates a two-phase search procedure consisting of two operations, QueryPPI(t_j)$\rightarrow \{p_i\}$ and AuthSearch($s, \{p_i\}, t_j$). This is illustrated in Figure 2. The first phase involves with the locator service in which the searcher poses query request, QueryPPI(t_j), and the PPI server returns a list of providers $\{p_i\}$ who may or may not have records of the requested owner t_j. The query evaluation in PPI server is trivial since the PPI, once constructed, contains the (obscured) mapping between providers and owners.

- AuthSearch(s, $\{p_i\}$, t_j): The second phase in the search is for searcher s to contact each provider in list $\{p_i\}$ (i.e. the result list from the first phase) and to find owner t_j's records there. This process involves user authentication and authorization regarding searcher s; we assume each provider has already set up its local access control subsystem for authorized access to the private personal records. Only after authorization

[3] To prevent every owner from setting the highest value of ϵ, a possible solution is to charge more when the owner sets a higher value of ϵ_j. It is reasonable since in a PPI system higher privacy settings comes with more search overhead.

can the searcher search the local repository on provider p_i.

We describe the internal data model in a PPI. Each personal record contains an owner identity t_j[4] (e.g. the person's name). As shown in Figure 3, a provider p_i summarizes its local record repository by a membership vector $M_i(\cdot)$; it indicates the list of owners who have delegated their records on provider p_i. For example, provider p_0 who has records of owner t_0 and t_1 maintains a membership vector as $M_i = \{t_0 : 1, t_1 : 1, t_2 : 0\}$. In our model, the same owner can have records spread across multiple providers (e.g., a patient can visit multiple hospitals). The constructed PPI maintains a mapping between providers and owners; it is essentially a combination of all provider-wise membership data, yet with noises. The PPI mapping data is an $m \times n$ matrix $M'(\cdot, \cdot)$, in which each row is of an owner, each column of a provider and each cell of a Boolean value to indicate the membership/non-membership of the owner to the provider. For the purpose of privacy preservation, there are noises or false positives added in the matrix; for example, regarding provider p_1 and owner t_0, value 1 in the published PPI M is a false positive in the sense that provider p_1 does not have any records of owner t_0 but falsely claims to do so. The false positive value is helpful for obscuring the true and private membership information.

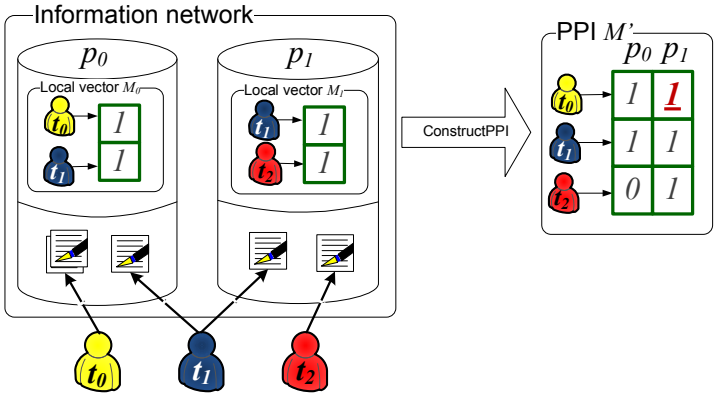


Figure 3: ϵ-PPI model

Table 1 summarizes the notations that will be used throughout the rest of the chapter.

[4]In this chapter, we use "owner" and "identity" interchangeably.

2.2.2 Threat Model

Privacy goals: In our work, we are mainly concerned with the owner-membership privacy; for an owner t_j, the owner-membership privacy is about which providers the owner t_j's records belong to, that is, $M(i, j) = 1$[5]. As mentioned, knowing this information, one can learn about private personal knowledge. Other privacy goals related to the PPI system but not addressed in this work include searcher anonymity and content privacy. The searcher anonymity prevents an attacker from knowing which owner(s) a searcher has searched for, which can be protected by various anonymity protocols [127]. The content privacy [45] involves the detailed content of an owner's record.

In order to attack the owner-membership privacy, we consider a threat model in which an attacker can exploit multiple information sources through different channels. In particular, we consider the following privacy-threatening scenarios:

- **Primary attack** : The primary attack scenario is that an attacker randomly or intentionally chooses a provider p_i and an owner t_j, and then claims that "owner t_j has delegated the records to provider p_i". To determine which providers and owners to attack, the attacker learns about the publicly available PPI data M', and attacks only those with $M'(i, j) = 1$. Given an owner t_j, the attacker can randomly or intentionally (e.g. by her prior knowledge) picks a provider p_i so that $M'(i, j) = 1$. To further refine the attack and improve the confidence, the attacker can exploit other knowledge through various channels, such as colluding providers. Due to space limit, we

[5]We use $M.(\cdot)$ and $M(\cdot, \cdot)$ interchangeably.

Table 1: Notations

Symbols of system model			
t_j	The j-th owner (identity)	n	Number of owners
ϵ_j	Privacy degree of t_j		
p_i	The i-th provider	m	Number of providers
$M_i(\cdot)$	Local vector of p_i	$M'(\cdot, \cdot)$	Data matrix in the PPI
Symbols of ϵ-PPI construction			
β_j	Publishing probability of t_j	σ_j	Frequency of owner t_j
λ	Percentage of common owners	fp_j	Achieved false positive rate of t_j

focus on the attack through the public channel in this chapter (the colluding attack and analysis can be found in the tech report [115]).

- **Common-identity attack** : This attack focuses on the common identity which appears in almost all providers in the network. The attacker can learn about the truthful frequency of owner identity σ_j from the public PPI matrix M' (as will be analyzed many PPI's [118, 45, 44] reveals the truthful frequency) and choose the owners with high frequency. By this means, the attacker can have better confidence in succeeding an attack. For example, consider the following extreme case: By learning an owner identity is with frequency $\sigma_j = 100\%$, the attacker can choose any provider and be sure that the chosen provider must be a true positive (i.e., $M(i,j) = 1$).

This chapter focuses on attacks on a single owner, while a multi-owner attack boils down to multiple single-owner attacks.

2.2.3 Privacy Metric and Degrees

Privacy metric: We measure the privacy disclosure by the attacker's confidence that the attack can succeed. Formally, given an attack on an owner t_j and provider p_i, we measure the privacy disclosure by the probability that the attack can succeed, that is, $Pr(M(i,j) = 1|M'(i,j) = 1)$. To measure the privacy protection level of a specific owner t_j, we use the average probability of successful attacks against all possible providers that are subject to $M'(i,j) = 1$. The privacy metric is formulated as following.

$$\begin{aligned} Pr(M(\cdot,j)|M'(\cdot,j)) &= \underset{\forall i, M'(i,j)=1}{\text{AVG}} \big(Pr(M(i,j) = 1|M'(i,j) = 1)\big) \\ &= 1 - fp_j \end{aligned}$$

Here, fp_j is the false positive rate of providers in the list of providers $M'(i,j) = 1$. The privacy disclosure metric on owner t_j is equal to $1 - fp_j$, because the false positive

providers determines the probability that an attack can succeed/fail. For example, if the list $\{p_i|M(i,j)=1\}$ is completely without any false positive providers (i.e. $fp_j = 0\%$), then attacks on any provider can succeed, leading to $100\% = 1 - fp_j$ success probability/confidence.

Based on the privacy metric, we further define four discrete privacy degrees. The definition of privacy degrees are based on an information flow model of our privacy threat model, in which an attacker obtains information from the information source through different channels.

- **UNLEAKED**: The information can not flow from the source (i.e. the original record delegated to a provider), and the attacker can not know the information. This is the highest privacy protection level.

- **ϵ-PRIVATE**: The information can flow to attackers through the channel of public PPI data or PPI construction process. If this occurs, the PPI design protects privacy from being disclosed. The PPI can provide a quantitative guarantee on the privacy leakage. Formally, given a privacy degree ϵ_j, it requires the quantitative guarantee as follows.

$$Pr(M(\cdot,j)|M'(\cdot,j)) \leq 1 - \epsilon_j \tag{1}$$

  In particular, when $\epsilon = 0\%$, the attacker might be 100% confident about a successful attack, and privacy is definitely leaked.

- **NOGUARANTEE**: The information can flow to the attacker and the PPI design can not provide any guarantee on privacy leakage. That is, the achieved value of privacy leakage metric may be unpredictable.

- **NOPROTECT**: The information can flow to the attacker and the PPI design does not address the privacy preservation. That is, the privacy is definitely leaked and the attack can succeed with 100% certainty. This is equivalent to the special case of NOGUARANTEE where $\epsilon_j = 0\%$. This is the lowest level of privacy preservation.

2.2.4 Analysis of Conventional PPIs

Based on our privacy model and metric, we analyze the privacy of existing PPI work and compare it with ϵ-PPI. Here, we consider the primary attack and the common-term attack . Before that, we briefly introduce the construction protocol of existing PPI. To be consistent with terminology, we use term to refer to owner's identity in this section, for example, the common-identity attack is referred to as the common-term attack.

Grouping PPI: Inspired by k-anonymity [112], existing PPI work [45, 44, 118] constructs its index by using a grouping approach. The idea is to assign the providers into disjoint privacy groups, so that true positive providers are mixed with the false positives in the same group and are made indistinguishable. Then, a group reports binary value 1 on a term t_j as long as there is at least one provider in this group who possesses the term. For example, consider terms are distributed in a raw matrix M as in Figure 3. If providers p_2 and p_3 are assigned to the same group, say g_1, then in the published PPI group $g_1$would report to have term t_0 and t_2 but not t_1, because both p_2 and p_3 do not have term t_1.

2.2.4.1 Privacy under primary attack

To form privacy groups, existing PPIs randomly assign providers to groups. By this means, the false positive rate resulted in the PPI varies non-deterministically. Furthermore, grouping based approach is fundamentally difficult to achieve per-term privacy degree. Because different terms share the same group assignment, even if one can tune grouping strategy (instead of doing it randomly) to meet privacy requirement for one or few terms, it would be extremely hard, if not impossible, to meet the privacy requirement for thousands of terms. For primary attack, the privacy leakage depends on the false positive rate of row at term t_j in PPI M'. This way, the grouping based PPI can at best provide a privacy level at NoGuarantee for primary attacks. Our experiments in Section 2.5.1.1 confirms our analysis as well.

2.2.4.2 *Privacy under common-term attack*

The grouping based PPI work may disclose the truthful term-to-provider distribution and thus the identity of common terms. We use a specific example to demonstrate this vulnerability.

An Example In an extreme scenario, one common term is with 100% frequency and all other terms show up in only one provider. For group assignment, as long as there are more than two groups, the rare terms can only show up in one group. In this case, the only common term in M' is the true one in M, in spite of the grouping strategy. This allows the attacker to be able to identify the true common terms in M and mount an attack against it with 100% confidence.

Given information of term distribution, one can fully exploit the vulnerability to amount common-term attacks. And the privacy degree depends on availability of term distribution information. For certain existing PPI [118], it directly leaks the sensitive common term's frequency σ_j to providers during index construction, leading to a NOPROTECT privacy level. Other PPI work, which does not leak exact term distribution information, still suffers from data-dependent privacy protection, resulting in a NOGUARANTEE privacy level.

We can summarize the analysis of prior work in Table 2.

Table 2: Comparison of ϵ-PPI against existing PPI's

	Primary attack	Common-identity attack
PPI [45, 44]	NOGUARANTEE	NOGUARANTEE
SS-PPI [118]	NOGUARANTEE	NOPROTECT
ϵ-PPI	ϵ-PRIVATE	ϵ-PRIVATE

2.2.5 Index Construction of Quantitative Privacy Preservation

In the ϵ-PPI, we aim at achieving ϵ-PRIVATE on a per-identity basis (i.e. personalizing privacy preservation for different owners). The formal problem that this chapter addresses is the index construction with quantitative privacy preservation, which is stated as below.

Proposition 2.2.1. *Consider a network with m providers and n owners; each provider p_i has a local Boolean vector M_i of its membership of n owners. Each owner t_j has a preferred level of privacy preservation ϵ_j. The problem of quantitatively personalized privacy preservation is to construct a PPI that can bound any attacker's confidence (measured by our per-owner privacy metric) under ϵ_j, with regards to all attacks on owner t_j as described in our threat model.*

2.3 ϵ-PPI Construction: Computation

Our ϵ-PPI construction is based on a proposed two-phase framework in which providers first collectively calculate a global value β, and then each provider independently publishes its local vector randomly based on probability β. This framework requires complex computations. In this section, we introduce them at different granularity: We first overview our two-phase construction framework and then introduce the first phase (called the β calculation) in details. At last, we conduct the privacy analysis.

2.3.1 A Two-Phrase Construction Framework

We propose a two-phase framework for the ϵ-PPI construction. First, for each owner identity t_j, all m providers collectively calculate a probability value β_j. In the second phase, the private membership value regarding owner t_j and every provider p_i is published. In this paragraph, we assume β_j is already calculated and we focus on describing the second phase – how to use β_j to publish private data. Recall that in our data model, each provider p_i has a Boolean value $M(i, j)$ that indicates the membership of owner t_j in this provider. After knowing value of β_j, provider p_i starts to publish this private Boolean value by randomly flipping it at probability β_j. To be specific, given a membership Boolean value (i.e. $M(i, j) = 1$), it is always truthfully published as 1, that is, $M'(i, j) = 1$. Given a non-membership value (i.e. $M(i, j) = 0$), it is negated to $M'(i, j) = 1$ at probability β_j. We call the negated value as the false positive in the published ϵ-PPI. The following formula describes the randomized publication. Note when Boolean value $M(i, j) = 1$, it is not

allowed to be published as $M'(i,j) = 0$.

$$\begin{aligned} 0 &\rightarrow \begin{cases} 1, \text{ with probability } \beta \\ 0, \text{ with probability } 1-\beta \end{cases} \\ 1 &\rightarrow 1 \end{aligned} \tag{2}$$

The truthful publication rule (i.e. $1 \rightarrow 1$) guarantees that relevant providers are always in the QueryPPI result and the 100% query recall is ensured. The false-positive publication rule (i.e. $0 \rightarrow 1$) adds noises or false positives to the published PPI which can help obscure the true owner-to-provider membership and thus preserves owner-membership privacy. For multiple owners, different β's are calculated and the randomized publication runs independently.

An example: Consider the case in Figure 3. For owner t_0, if β_0 is calculated to be 0.5, then provider p_1 would publish its negative membership value $M_1(0) = 0$ as value 1 with probability 0.5. In this example, it is flipped and the constructed ϵ-PPI contains $M'(1,0) = 1$. Similarly for identity t_2 and provider p_0, it is also subject to flipping at probability β_2. In this example, it is not flipped, and the constructed ϵ-PPI contains $M'(0,2) = 0$.

2.3.2 The β Calculation

In the randomized publication, β_j determines the amount of false positives in the published ϵ-PPI. For quantitative privacy preservation, it is essential to calculate a β_j value that makes the false positive amount meet the privacy requirement regarding ϵ_j. In this part, we focus on describing the calculation of β which serves as the first phase in ϵ-PPI construction process. Concretely we consider two cases: the common identity case and the non-common identity case. Recall that the common identity refers to such an owner who delegates her records to almost all providers in the network. The general PPI construction is vulnerable to the common-identity attack and it needs to be specially treated.

2.3.2.1 The Case of Non-common Identity

In the case of non-common identity, negative providers suffice to meet the desired privacy degree. We consider the problem of setting value β_j for identity t_j in order to meet the desired ϵ_j. Recall the randomized publication: Multiple providers independently runs an identical random process, and this can be modeled as a series of Bernoulli trials (note that the publishing probability $\beta(t_j)$ is the same to all providers). Our goal is to achieve privacy requirement that $fp_j \geq \epsilon_j$ with high level success ratio p_p, that is, $p_p = Pr(fp_j \geq \epsilon_j)$. Under this model, we propose three policies to calculate β with different quantitative guarantees: a basic policy β_b that guarantees $fp_j \geq \epsilon_j$ with 50% probability, and an incremented expectation based policy β_d, and a Chernoff bound based policy β_c that guarantees $fp_j \geq \epsilon_j$ with γ probability where success ratio γ can be configured.

Basic policy: The basic policy sets the β value so that the expected amount of false positives among m providers is satisfactory, that is, be at least $\epsilon_j \cdot m(1-\sigma_j)$. Formally,

$$\begin{aligned} \epsilon_j &= \frac{(1-\sigma_j)\cdot\beta_b(t_j)}{(1-\sigma_j)\cdot\beta_b(t_j)+\sigma_j} \\ \Rightarrow \beta_b(t_j) &= [(\sigma_j^{-1}-1)(\epsilon_j^{-1}-1)]^{-1} \end{aligned} \tag{3}$$

The basic policy has poor quality in attaining the desired privacy preservation; the actual value fp_j is bigger than ϵ_j with only 50% success ratio.

Incremented expectation-based policy: The incremented expectation-based approach is to increase the expectation-based $\beta_b(t_j)$ by a constant value, that is,

$$\beta_d(t_j) = \beta_b(t_j) + \Delta \tag{4}$$

Incremental Δ can be configurable based on the quality requirement; the bigger the value is, the higher success ratio p_p is expected to attain. However, there is no direct connection between the configured value of Δ and the success ratio p_p that can be achieved, leaving it a hard task to figure out the right value of Δ based on desired p_p.

Chernoff bound-based policy: Toward an effective policy to calculate β, we apply the Chernoff bounds to the randomized publication process which is modeled as Bernoulli trials. This policy allows direct control of the success ratio. Formally, it has the property described in Theorem 2.3.1.

Theorem 2.3.1. *Given desired success ratio $\gamma > 50\%$, let $G_j = \frac{\ln \frac{1}{1-\gamma}}{(1-\sigma_j)m}$ and*

$$\beta_c(t_j) \geq \beta_b(t_j) + G_j + \sqrt{G_j^2 + 2\beta_b(t_j)G_j} \tag{5}$$

Then, randomized publication with $\beta(t_j) = \beta_c(t_j)$ statistically guarantees that the published ϵ-PPI can meet privacy requirement $fp_j \geq \epsilon_j$ with success ratio larger than γ.

Proof. We model the problem as Bernoulli trials and prove the theorem by applying Chernoff bound. For a term t_j, the total number of false positive providers is modeled as the sum of $T = m(1-\sigma_j)$ Bernoulli trials, because there are $m(1-\sigma_j)$ negative providers for term t_j and each negative provider independently and randomly publishes its own bit, a process that can be modeled as a single Bernoulli trials. In the trial, when the negative provider becomes a false positive (i.e., $0 \rightarrow 1$) which occurs at probability $\beta(t_j)$, the Bernoulli random variable, denoted by X, takes on value 1. Otherwise, it takes the value 0. Let $E(X)$ be the expectation of variable X, which in our case is,

$$E(X) = m(1-\sigma_j) \cdot \beta(t_j) \tag{6}$$

We can apply the Chernoff bound for the sum of Bernoulli trials, $Pr(X \leq (1-\delta)E(X)) \leq e^{-\delta^2 E(X)/2}$ [96], where $\delta > 0$ is any positive number. For term t_j, the expected success rate, denoted by $p_p(t_j)$, is equal to the probability of a publication success, that is, $p_p(t_j) = Pr(fp_j > \epsilon_j)$. Note $fp_j = \frac{X}{X+\sigma_j \cdot m}$, we have,

$$\begin{aligned} p_p(t_j) &= 1 - Pr(fp_j \leq \epsilon_j) \\ &= 1 - Pr(X \leq m\frac{\sigma_j}{\epsilon_j^{-1} - 1}) \\ &\geq 1 - e^{-\delta_j^2 m(1-\sigma_j)\beta(t_j)/2} \end{aligned} \tag{7}$$

In here, $\delta_j = 1 - \frac{1}{(\epsilon_j^{-1}-1)(\sigma_j^{-1}-1)} \cdot \frac{1}{\beta(t_j)} = 1 - \frac{\beta_b(t_j)}{\beta(t_j)}$. Recall that γ is the required minimal success rate. If we can have

$$1 - e^{-\delta_j^2 m(1-\sigma_j)\beta(t_j)/2} \geq \gamma \tag{8}$$

for all indexed terms, then $\forall j, p_p(t_j) \geq \gamma$. This means in the case of large number of terms, the percentage of successfully published terms or p_p is expected to be larger than or equal to γ, i.e., $p_p \geq \gamma$, which confirms the proposition. Hence, by plugging δ_j in Equation 8, we can derive,

$$(\beta_c(t_j))^2 - 2\Big(\beta_b(t_j) + \frac{\ln \frac{1}{1-\gamma}}{(1-\sigma_j)m}\Big)\beta_c(t_j) + (\beta_b(t_j))^2 \geq 0$$

Note $\frac{\ln \frac{1}{1-\gamma}}{(1-\sigma_j)m} = G_j$, and $\beta_c(t_j)$ should be bigger than $\beta_b(t_j)$ since success ratio is larger than 50%. Solving the inequality and taking only the solution that satisfies $\beta_c(t_j) > \beta_b(t_j)$, we have,

$$\beta_c(t_j) \geq \beta_b(t_j) + G_j + \sqrt{G_j^2 + 2\beta_b(t_j)G_j}$$

□

2.3.2.2 The Case of Common Identities

With the above β calculation for non-common identities, the constructed ϵ-PPI is vulnerable to the common-identity attack. Because the β_*[6] bears information of identity frequency σ_j, and during our index construction framework, β needs to be released to all participating providers. A colluding provider would release such information to the attacker who can easily obtain the truthful identity frequency σ (e.g., from Equation 3 assuming ϵ_j is publicly known) and effectively formulates the common-identity attack.

To defend against the common-identity attack, ϵ-PPI construction employs an identity-mixing technique for common identities. The idea is to mix common identities with certain non-common identities by exaggerating the calculated β_j (i.e. falsely increasing certain

[6]We use β_* to denote the probability value calculated by any of the three policies for non-common identities.

β_j to 100%) from which one can not distinguish common identities from the rest. To be specific, for a non-common identity t_j, we allow its β_j to be exaggerated to 100% with probability λ, that is,

$$\beta = \begin{cases} \begin{cases} \beta_*, & 1-\lambda \\ 1, & \lambda \end{cases}, & \beta_* < 1 \\ 1, & \beta_* \geq 1 \end{cases} \tag{9}$$

Given a set of common identities, we need to determine how many non-common identities should be chosen for mixing, in other words, to determine the value of λ. While a big value of λ can hide common identities among the non-common ones, it incurs unnecessarily high search cost. On the other hand, a value of λ which is too small would leave common identities unprotected and vulnerable. In ϵ-PPI, we use the following heuristic-based policy to calculate λ.

- In the set of mixed identities, the percentage of non-common identities should be no smaller than ξ. Since there are $\sum_{\beta_* \geq 1} 1$ common identities and thus $\sum_{\beta_* < 1} \lambda$ non-common identities in the set, we have the following formula.

$$\begin{aligned} \xi &\leq \frac{\sum_{\beta_* < 1} \lambda}{\sum_{\beta_* \geq 1} 1 + \sum_{\beta_* < 1} \lambda} \\ \Rightarrow \lambda &\geq \frac{\xi}{1-\xi} \cdot \frac{\sum_{\beta_* \geq 1} 1}{n - \sum_{\beta_* \geq 1} 1} \end{aligned} \tag{10}$$

2.3.2.3 β Calculation: Putting It Together

We summarize the β calculation in the ϵ-PPI construction. For each identity t_j, $\beta(t_j)$ is calculated based on Equation 9, which follows the computation flows as below. The underline symbol indicates the variable is private and $\Rightarrow$ indicates the computation is fairly complex and heavy (e.g. involving square root when calculating β_*).

$$\begin{aligned} &\text{Frequency } \underline{\sigma} \Rightarrow \text{Raw probability } \underline{\beta_*} \rightarrow \\ &\rightarrow \textstyle\sum_{\underline{\beta_* \geq 1}} 1 \rightarrow \text{Common id percentage } \lambda \rightarrow \text{Final probability } \beta \end{aligned} \tag{11}$$

2.3.3 Privacy Analysis of Constructed ϵ-PPI

We present the privacy analysis of the constructed ϵ-PPI under our threat model.

Privacy under primary attack: The property of the three policies of calculating β_* suggests that the false positive rate in the published ϵ-PPI should be no smaller than ϵ_j in a statistical sense. Recall that the false positive rate bounds the attacker's confidence; it implies that ϵ-PPI achieves an ϵ-PRIVATE degree against the primary attack. It is noteworthy that our ϵ-PPI is fully resistant to repeated attacks against the same identity over time, because the ϵ-PPI is static; once constructed and having privacy protected, it stays the same.

Privacy under common-identity attack: For the common-identity attack, the attacker's confidence in choosing a true common identity depends on the percentage of true common identities among the (mixed) common identities in the published ϵ-PPI. Therefore the privacy preservation degree is bounded by the percentage of false positives (in this case, it depends on the percentage of the non-common identities which is mixed and published as common identities in the published ϵ-PPI), which equals ξ. By properly setting λ, we can have $\xi = \max_{\forall t_j \in \{\text{common identities}\}} \epsilon_j$. By this way, it is guaranteed to achieve the per-identity ϵ-PRIVATE degree against the common-identity attack.

2.4 ϵ-PPI Construction: Realization

This section describes the design and implementation of a distributed and secure protocol that realizes the computation of ϵ-PPI construction described in the previous section.

2.4.1 Challenge and Design

The goal of our protocol is to efficiently and securely compute the publishing probability $\{\beta_j\}$ among a set of mutually untrusted providers who are reluctant to exchange the private membership vector with others. The computation is based secure multi-party computation (or MPC) that protects the input-data privacy. It is challenging to construct ϵ-PPIusing

MPC in a large information network. On the one hand, current techniques for MPC only support small computation workloads [97]. On the other hand, the computation required in ϵ-PPI construction is big and complex; the computation model involves large number of identities and providers; even for a single identity it involves fairly complex computation (e.g., square root and logarithm as in Equation 5). This poses a huge challenge to design a practical protocol for secure ϵ-PPI construction.

To address the above challenge, we propose an efficient and secure construction protocol based on the principle of *minimizing the secure computation*. Given a computation flow in Equation 11, our protocol design has three salient features: 1) It separates the secure and non-secure computations by the last appearance of private variables in the flow (note that the computation flows from the private data input to the end of non-private result). 2) It reorders the computation to minimize the expensive secure computation. The idea is to push down complex computation towards the non-private end. To be specific, instead of first carrying out complex floating point computations for raw probability β, as in Formula 11, we push such computations down through the flow and pull up the obscuring computations for private input, as in Formula 12. 3) To scale to a large number of providers, we propose an efficient protocol for calculating the secure sum, and use it to reduce the "core" of the MPC part.

$$\underline{\sigma} \rightarrow \quad \sum_{\underline{\sigma}<\sigma'} 1 \quad \rightarrow \lambda \rightarrow \begin{cases} \rightarrow & \beta = 1 \\ \Rightarrow & \beta = \beta_* \end{cases} \tag{12}$$

2.4.2 The Distributed Algorithm

Following our design, we propose a practical distributed algorithm to run the two-phase ϵ-PPI construction. The overall workflow is illustrated in Figure 4. For simplicity, we focus on phase 1 for β calculation. The β calculation is realized in two stages by itself: As illustrated in Algorithm 1, the first stage is a SecSumShare protocol which, given m input Boolean from the providers, outputs c secret shares whose sum is equal to the sum

of these m Boolean. Here, c is the number of shares that can be configurable based on the tolerance on provider collusion. The output c shares have the security property that a party knowing $x < c$ shares can not deduce any information about the sensitive sum of m Boolean. For different identities, the SecSumShare protocol runs multiple instances independently and in parallel, which collectively produce c vectors of shares, denoted by $s(i, \cdot)$, where $i \in [0, c-1]$. The c vectors are distributed to c coordinate providers (for simplicity we assume they are providers $p_0, \ldots, p_{c-1}$) on which the second-stage protocol, CountBelow, is run. As shown by Algorithm 2, given c vectors $s(0, \cdot), \ldots s(c-1, \cdot)$ and a threshold t, the CountBelow algorithm sums them to vector $\sum_i s(i, \cdot)$ and counts the number of elements that are bigger than t.

Table 3: Distributed algorithms for ϵ-PPI construction

Algorithm 1 calculate-beta($M_0, \ldots M_{n-1}$)

1: $\{s(0, \cdot), \ldots s(c-1, \cdot)\} \leftarrow$ SecSumShare($M_0, \ldots M_{n-1}$)
2: $\sigma'(\cdot)$ is calculated under condition $\beta_* = 1$, by either Equation 3, or 4 or 5.
3: $\sum_{\sigma \geq \sigma'} 1 \leftarrow$CountBelow($s(0, \cdot), \ldots s(c-1, \cdot), \sigma'(\cdot) \cdot m$)
4: $\{\beta_0, \ldots \beta_{m-1}\} \leftarrow \sum_{\sigma \geq \sigma'} 1$ ▷ By Equation 12

Algorithm 2 CountBelow($s(0, \cdot), \ldots s(c-1, \cdot)$, threshold t)

count$\leftarrow 0$
for $\forall j \in [0, m-1)$ **do**
    $S[j] \leftarrow \sum_i s(i, j)$
    if $S[j] < t$ **then**
        count++
    end if
end for
return count

2.4.2.1 Distributed Algorithm for SecSumShare

We use an example in the top box in Figure 4 to illustrate the distributed algorithm of SecSumShare. In the example $c = 3$ and there are five providers $p_0, \ldots p_4$. The example focuses on a single identity case for t_j (e.g. $j = 0$). Out of the 5 providers, p_1 and p_2 have records of owner t_0 (i.e., $M(1, 0) = M(2, 0) = 1$). SecSumShare requires modular operations; in this example, the modulus divisor is $q = 5$. It runs in the following 4 steps.

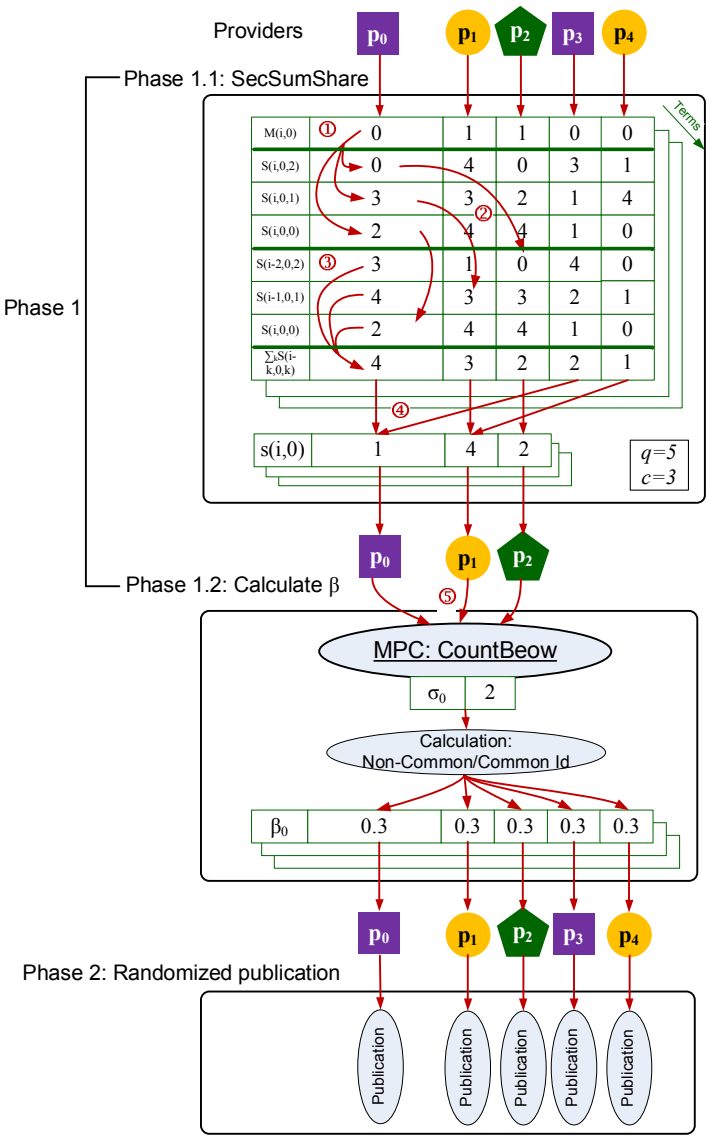


Figure 4: An example of ϵ-PPI construction algorithm

1 Generating shares: each provider p_i decomposes its private input Boolean $M(i,j)$ into c shares, denoted by $\{S(i,j,k)\}$, with $k \in [0, c-1]$. The first $c-1$ shares are randomly picked from interval $[0,q]$ and the last share is deterministically chosen so that the sum of all shares equals the input Boolean $M(i,0)$ in modulo q. That is, $(\sum_{k\in[0,c]} S(i,j,k)) \mod q = M(i,j)$. In Figure 4, as depicted by arrows ①, p_0's input $M(0,0)$ is decomposed to $c=3$ shares, $\{S(0,0,k)|k\} = \{2,3,0\}$. It ensures $(2+3+0) \mod 5 = 0$.

2 Distributing shares: each provider p_i then distributes its shares to the next $c-1$ neighbor providers; k-th shares $S(i,j,k)$ will be sent out to k-th successor of provider p_i, that is, $p_{(i+k) \mod m}$. As shown by arrows ② in Figure 4, p_0 keeps the first share 2 locally, sends its second share 3 to its successor p_1 and the third share 0 to its 2-hop successor p_2.

3 Summing shares: each provider then sums up all shares she has received in the previous step to obtain the *super-share*. In Figure 4, after the step of share distribution, provider p_0 receives 3 from p_3, 4 from p_4 and 2 from herself. As depicted by arrows ③, the super-share is calculated to be $3+4+2 \mod 5 = 4$.

4 Aggregating super-shares: each provider sends its super-share to a set of c coordinators. These coordinators receiving super-shares then sum the received shares up and output the summed vector $s(i,\cdot)$ to the next-stage CountBelow protocol. In Figure 4, provider p_0, p_1, p_2 are chosen as coordinators and arrow ④ shows that the sum of super-shares on provider p_0 is $s(0,0) = (4+2) \mod 5 = 1$. The sum of all the values on coordinators should be equal to the number of total appearances of identity t_0. That is, $1+4+2 \mod 5 = 2$. Note that there are two providers with identity t_0. This total appearance number or identity frequency may be sensitive (in the case of common identity) and can not be disclosed immediately, which is why we need the second stage protocol, CountBelow.

2.4.2.2 Implementation of CountBelow computation

The secure computation of CountBelow (in Algorithm 2) is implemented by using a generic MPC protocol. Each party corresponds to a coordinate provider in the ϵ-PPI system. Specifically, we choose a Boolean-circuit based MPC protocol FairplayMP [47] for implementation. Since Algorithm 2 is implemented by expensive MPC it normally becomes the bottleneck of the system; in practice, $c \ll m$ and thus the network can scale to large number of providers m while the MPC is still limited to a subset of the network.

2.4.3 Privacy Analysis of Constructing ϵ-PPI

We analyze the privacy preservation of ϵ-PPI construction process. We mainly consider a semi-honest model, which is consistent with the existing MPC work [47]. The privacy analysis is conducted from three aspects: 1) The privacy guarantee of SecSumShare protocol. It guarantees: 1.1) $(2c-3)$-secrecy of input privacy [118]: With less than c providers in collusion, none of any private input can be learned by providers other than its owner. 1.2) c-secrecy of output privacy: The private sum can only be reconstructed when all c shares are used. With less than c shares, one can learn nothing regarding the private sum. The output privacy is formally presented in Theorem 2.4.1. 2) The security and privacy of CountBelow relies on that of the MPC used in implementation. The generic MPC technique can provide information confidentiality against c colluding providers [47]. 3) The final output β does not carry any private information, and is safe to be released to the (potentially untrusted) providers for the randomized publication.

Theorem 2.4.1. *The SecSumShare's output is a (c, c) secret sharing scheme. Specifically, for an owner t_j, SecSumShare protocol outputs c shares, $\{s(i,j)|\forall i \in [0, c-1]\}$, whose sum is the secret v_j. The c shares have the following properties.*

- Recoverability*: Given c output shares, the secret value v_j (i.e. the sum) can be easily reconstructed.*
- Secrecy*: Given any $c-1$ or fewer output shares, one can learn nothing about the secret value, in the sense that the value's conditional distribution given the known shares is the same as the prior distribution,*

$$\forall x \in \mathbb{Z}_q, Pr(v_j = x) = Pr(v_j = x | V \subset \{s(i,j)\})$$

where V is any proper subset of $\{s(i,j)\}$.

Proof. Recoverability can be trivially proved based on the fact that $\sum_{\forall i \in [0,c-1]} s(i,j) = v_j$.

To prove secrecy, we examine the process of generating super-shares $s(i, j)$. It is easy to see that the SecSumShare protocol uses a (c, c) secret sharing to split each private input $M(i, j)$. The generated c shares for each input value are distributed to c different output super-shares. For each private input $M(i, j)$, an output super share $s(i, j)$ has included *one and only one* share from it. Therefore, when an adversary knows at most $c - 1$ outputs, at least one share of each private input is still unknown to her. This leaves the value of any input completely undetermined to this adversary, thus the secret or the sum of input values completely undetermined. □

2.5 Experiments

To evaluate the proposed ϵ-PPI, we have done two set of experiments: The first set, based on simulations, evaluates how effective the ϵ-PPI can be in terms of delivering quantitative privacy protection. The second set evaluates the performance of our index construction protocol. For realistic performance results, we have implemented a functioning prototype for ϵ-PPI construction.

2.5.1 Effectiveness of Privacy Preservation

Experimental setup: To simulate the information network, we use a distributed document dataset [94] of $2,500 - 25,000$ small digital libraries, each of which simulates a provider in our problem setting. To be specific, this dataset defines a "collection" table, which maintains the mapping from the documents to collections. The documents are further derived from NIST's publicly available TREC-WT10g dataset [77]. To adapt to our problem setting, each collection is treated as a provider and the source web URLs (as defined in TREC-WT10g dataset) of the documents are treated as owner's identity. If not otherwise specified, we use no more than $10,000$ providers in the experiments. Using the collection table, it also allows us to emulate the membership matrix M. The dataset does not have a privacy metric for the query phrase. In our experiment, we randomly generate the privacy degree ϵ in the domain $[0, 1]$. We use a metric, success ratio, to measure the

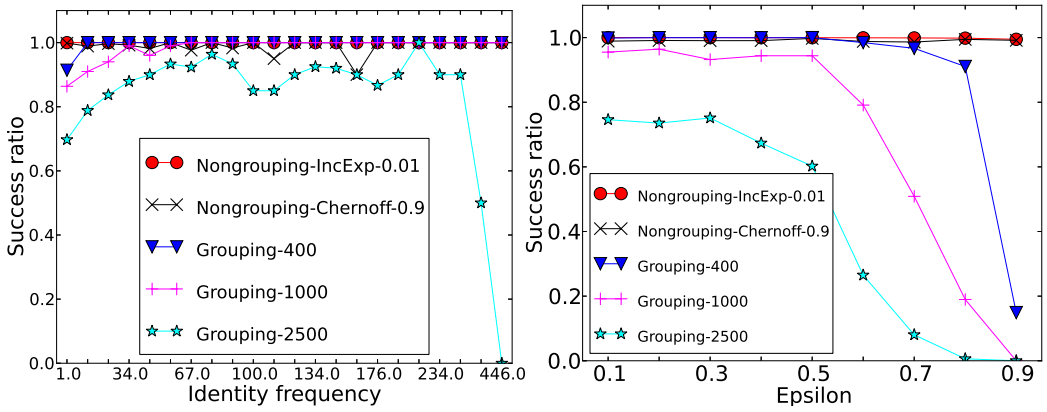
effectiveness. The success ratio is the percentage of identities whose false positive rates in the constructed PPI are no smaller than the desired rate ϵ_j.

2.5.1.1 ϵ-PPI versus Existing Grouping-based PPI's

The experiments compare ϵ-PPI with existing PPI's. The existing PPI's [45, 44, 118] are based on a grouping technique; providers are organized into disjoint privacy groups so that different providers from the same group are indistinguishable from the searchers. By contrast, ϵ-PPI does not utilize grouping technique and is referred to in this section as a non-grouping approach. In the experiment, we measure the success ratio of privacy preservation, and search performance. Grouping PPI's are tested under different group sizes. Given a network of fixed providers, we use the group number to change average group size. We test grouping PPI with the Chernoff bound-based and the incremented expectation-based policies under the default setting. The expected false positive rate is configured at 0.8, and the number of providers is $10,000$. We uniformly sample 20 times and report the average results.

Results are illustrated in Figure 5. Non-grouping PPI generally performs much better and more stable than the grouping approach in terms of success ratio. With proper configuration (e.g. $\Delta = 0.01$ for incremental expectation-based policy and $\gamma = 0.9$ for Chernoff policy), the non-grouping ϵ-PPI always achieves near-optimal success ratio (i.e. 1.0). By contrast, the grouping PPI's display instability in their success ratio. For example, as shown by the "Grouping ($\#$groups 2000)" series in Figure 5a, the success ratio fluctuates between 0.0 and 1.0, which renders it difficult to provide a guarantee to the system and owners. The reason is that with 2000 groups, sample space in each group is too small (i.e., with 50 providers) to hold a stable result for success ratio. When varying ϵ, similar behavior is shown in Figure 5b; the success ratio of grouping PPI's quickly degrades to 0, leading to unacceptable privacy quality. This is due to the grouping design of PPI that is agnostic to different owners. This set of experiments shows that the privacy degree of non-grouping

PPI's can be effectively tuned, implying the ease of a practical use.



(a) Varying identity frequency

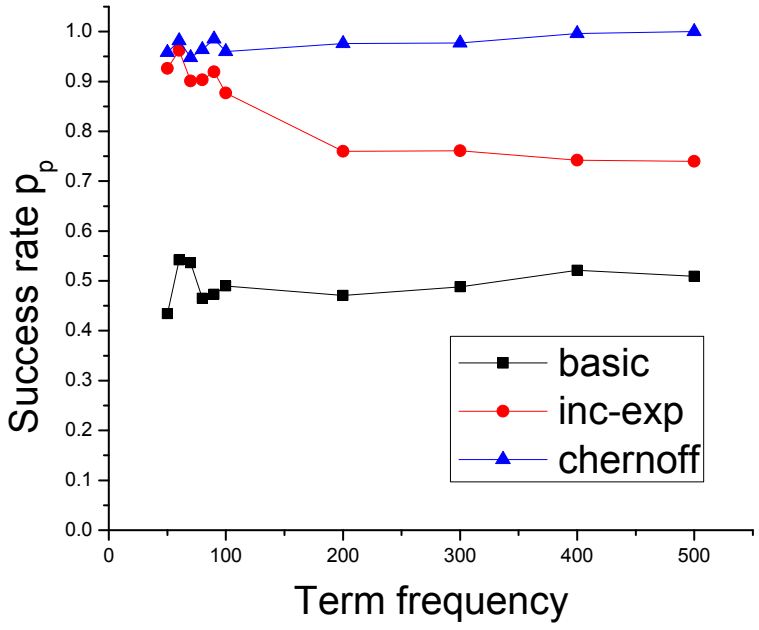
(b) Varying ϵ

Figure 5: Comparing non-grouping and grouping

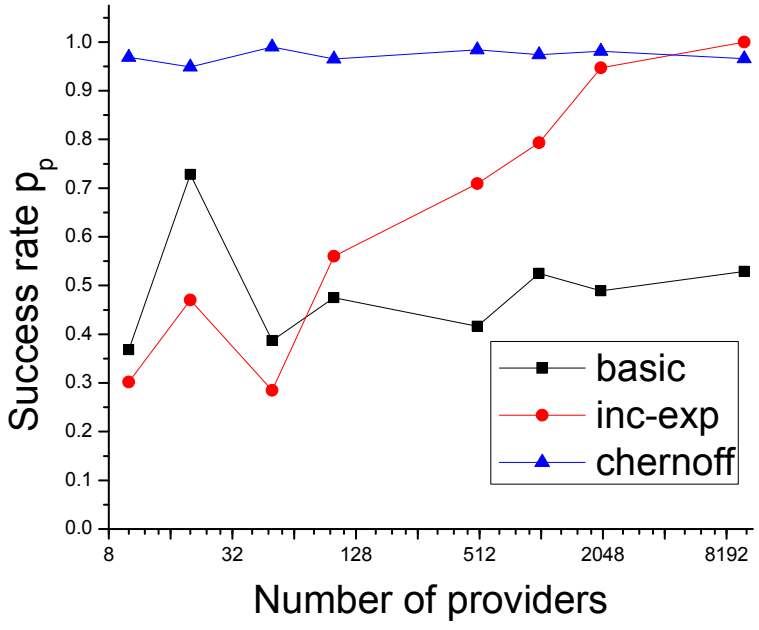
2.5.1.2 *Comparing different β-calculation policies*

We evaluate and compare the effectiveness of three β-calculation policies with ϵ-PPI. In the experiments, we tested various parameter settings and show the representative results at the following settings: $\Delta = 0.02$ as in the incremented expectation-based policy, expected success ratio $\gamma = 0.9$ as in the Chernoff bound based policy. The default false positive rate is set at $\epsilon = 0.5$. The experiment results measuring the success ratio are reported in Figure 6. In Figure 6a, we vary the identity frequency from near 0 to about 500 providers with totally $10,000$ providers in the network. In Figure 6b we vary the number of providers and set the identity frequency to be constant 0.1. It can be seen from the results that while the Chernoff bound-based policy (with $\gamma = 0.9$) always achieves near-optimal success ratio (i.e., close to 1.0), the other two policies fall short in certain cases; the expectation-based policy is not configurable and achieves the success rate only at 0.5. This is expected because the expectation-based approach works on an average sense. For the incremented expectation-based policy, its success ratio, though approaching 1.0 in some cases, is much smaller than 1.0 and unsatisfactory in other cases (e.g. for terms of high frequency as in Figure 6a and for few providers as in Figure 6b). On the other hand, the high-level privacy

preservation of the Chernoff bound policy comes with reasonable search overhead. The related experiment results can be found in technical report [115].



(a) Varying frequency under $10,000$ providers



(b) Varying provider numbers under frequency 0.1

Figure 6: Quality of privacy preservation

2.5.2 Performance of Index Construction

Experimental setup: We evaluate the performance of our distributed ϵ-PPI construction protocol. Towards that, we have implemented a functioning prototype. The CountBelow is implemented by using an MPC software, FairplayMP [47], which is based on Boolean circuits. The implemented CountBelow protocol is written in SFDL, a secure function definition language exposed by FairplayMP, and is compiled by the FairplayMP runtime to Java code, which embodies the generated circuit for secure computation. We implement the SecSumShare protocol in Java. In particular, we use a third-party library Netty [22] for network communications and Google's protocol buffer [30] for object serialization. We conduct experiments on a number of machines in Emulab [18, 126], each equipped with a 2.4 GHz 64-bit Quad Core Xeon processor and 12 GB RAM. In the experiments, the number of machines tested is varied from 3 to 9 (due to limited resource at hand). For each experiment, the protocol is compiled to and run on the same number of parties. Each party is mapped to one dedicated physical machine. The experiment uses a configuration of $c = 3$.

To justify the standpoint of our design that MPC is expensive, we compare our MPC-reduced approach as in the ϵ-PPI construction protocol against a pure MPC approach. The pure MPC approach does not use the SecSumShare protocol to reduce the number of parties in the generic MPC part and directly accepts inputs from the m providers. The metric used in the experiment is the start-to-end execution time, which is the time duration from when the protocol starts to run to when the last machine reports to finish. The result is shown as in Figure 7a. It can be seen that the pure MPC approach generally incurs longer execution time than our MPC-reduced approach (used in ϵ-PPI construction): As the information network grows large, while the execution time of pure MPC approach increases super-linearly, that of MPC-reduced approach increases slowly. This difference is due to the fact that the MPC in our MPC-reduced approach is fixed to c parties and does

not change as the number of providers m grows. And the parallel SecSumShare in MPC-reduced approach is scalable in m as well, since each party runs in constant rounds, and each round sends a constant number (at most $c-1$) of messages to its neighbors. For scaling with more parties, we use the metric of circuit size, which is the size of the compiled MPC program. As a valid metric, the circuit size determines the execution time[7] in real runs. By this means, we can show the scalability result of up to 60 parties as in Figure 7b. Similar performance improvement can be observed except that the circuit size grows linearly with the number of parties involved. Finally, we also study the scalability from running the protocol with multiple identities in a three-party network. The result in Figure 7c shows that ϵ-PPI construction grows with the number of identities at a much slower rate than that of the pure MPC approach.

2.6 Related Work

2.6.1 Privacy-Preserving Data Indexing

Non-encryption based PPI: PPI is designed to index access controlled contents scattered across multiple content providers. While being stored on an untrusted server, PPI aims at preserving the content privacy of all participant providers. Inspired by the privacy definition of k-anonymity [112], existing PPI work [45, 44, 118] follows the *grouping-based* approach; it organizes providers into disjoint privacy groups, such that providers from the same group are indistinguishable to the searchers. To construct such indexes, many existing approaches [45, 44, 46] assume providers are willing to disclose their private local indexes, an unrealistic assumption when there is a lack of mutual trust between providers. SS-PPI [118] is proposed with resistance against colluding attacks. While most existing grouping PPI's utilize a randomized approach to form groups, its weakness is studied in SS-PPI but without a viable solution. Though the group size can be used to

[7] Regarding the detailed definition of circuit size and the exact correlation between circuit size and execution time, it can be found in FairplayMP [47].

configure grouping-based PPI's, it lacks per-owner concerns and quantitative privacy guarantees. Moreover, organizing providers in groups usually leads to query broadcasting (e.g, with positive providers scattered in all groups), rendering search performance inefficient. By contrast, ϵ-PPI is a brand new PPI abstraction without grouping (i.e. non-grouping PPI as mentioned before), which provides quantitative privacy control on a per-owner basis.

Index with search-able encryption: Building search-able indexes over encrypted data has been widely studied in the context of both symmetric key cryptography [109] and public key cryptography [125, 92, 51]. In this architecture, content providers build their local indices and encrypt all the data and indices before submitting them to the untrusted server. During query time, the searcher first gets authenticated and authorized by the corresponding content provider; the searcher then contacts the untrusted server and searches against the encrypted index. This system architecture makes the assumption that a searcher already knows which provider possesses the data of her interest, which is unrealistic in the PPI scenario. Besides, unlike the encryption-based system, performance is a motivating factor behind the design of our PPI, by making no use of encryption during the query serving time.

2.6.2 Secure Distributed Computations

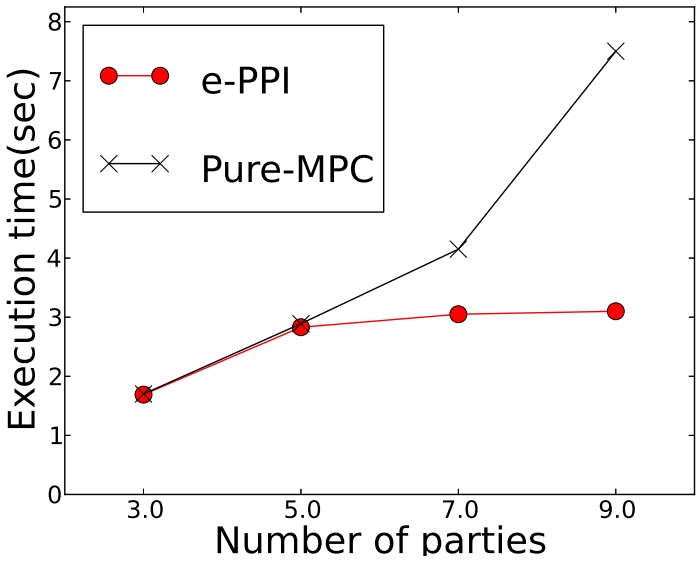
Practical MPC: Recently a large body of research work [95, 47, 79, 60, 35] is dedicated towards a practical MPC platform. Traditional work for generic MPC largely falls under two models, the garbled functions used for Boolean circuits and the homomorphic encryption used for arithmetic calculation. Towards efficient and practical MPC systems, Fairplay [95, 47] implements the computation of Boolean circuits for two or more parties, and VIFF [60] is a runtime for the computation of arithmetic circuits. MightBeEvil [80] supports efficient two-party computation via garbled circuit. Based on the observation that different computation models can lead to performance gain for different workloads,

TASTY [79] proposes to use a modular and adaptive design which divides the whole workload into several modules and accordingly maps the modular workload to to a specific MPC model. It is realized by a scheme that can convert the encrypted or garbled data between modules. Recent work [35] extends the domain of practical MPC (from integers) to floating point numbers by using Shamir's secret sharing.

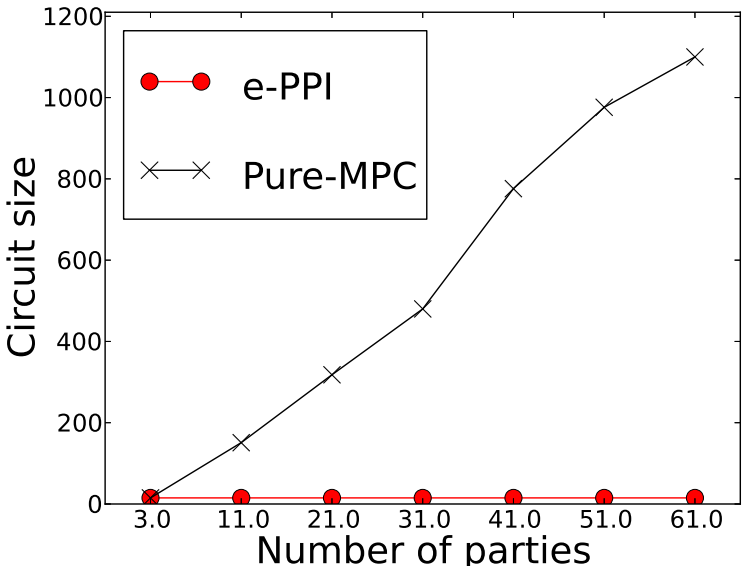
Privacy-preserving multi-source analysis: Based on the primitive provided by MPC, there is a large body of research work on privacy-preserving analysis of multi-source data. We briefly survey this research area. Private record linkage (or PRL) [64, 98] is important in the health information exchanges. The task of PRL is to identify medical records of the same patient which are with semantically heterogeneous demographic information and are distributed across multiple Healthcare providers. In industry, Master Patient Indexes [23, 25] are real-world systems that recently emerge to support PRL. In the research community, privacy-preserving PRL schemes are recently proposed [89, 90]. The PRL technique is complementary to our ϵ-PPI in the sense that they could work together to support a federated search service for patient medical history based on heterogeneously distributed patient data. In the domain of corporate IT business, prior work [58] proposes a privacy-preserving framework for analyzing OLAP workloads for business intelligence (or BI). For data sharing between multiple corporate entities, DJoin [97] is proposed for privacy-preserving join computation. To address the inefficiency of generic MPC, DJoin uses an efficient but domain-specific primitive for set operations [86] in combination with MPC. In similar spirit, our ϵ-PPI construction protocol reduces the expensive MPC by applying the efficient secure-sharing technique.

2.7 Conclusion

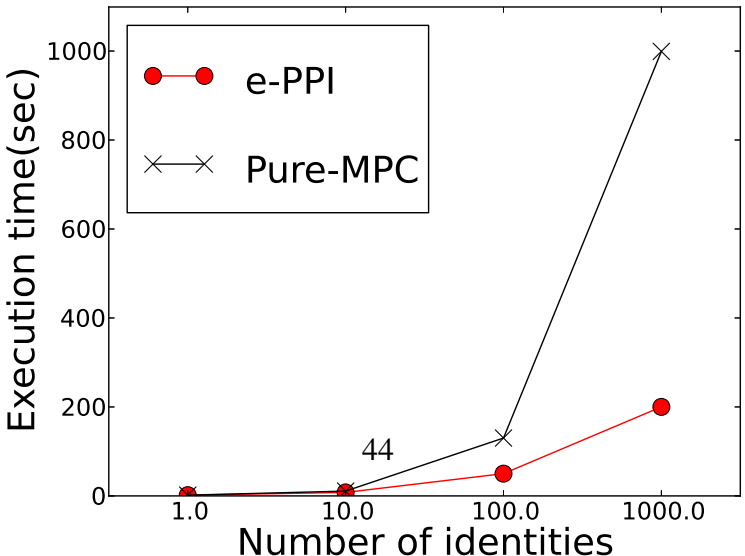
In this chapter, we propose ϵ-PPI for personalized privacy control with quantitative guarantee. ϵ-PPI allows each data owner to specify her personal preference in privacy preservation, and ϵ-PPI can guarantee that such personalized privacy can be preserved in a quantitative fashion. In the design of ϵ-PPI, we identify a vulnerability of generic PPI systems, the common-identity attack. We propose an identity-mixing mechanism to protect ϵ-PPI against such attacks. We have implemented the construction protocol for ϵ-PPI without any trusted party involved. We optimize the performance of secure index construction protocol by minimizing the use of expensive MPC. We have built a generic privacy threat model and performed security analysis which shows the advantages of ϵ-PPI over other PPI system in terms of privacy preservation quality.



(a) Execution time with single identity



(b) Circuit size with single identity



CHAPTER III

HINDEX: SUPPORTING BIG-DATA INDEX ON SCALABLE KEY-VALUE STORES IN A CLOUD

Big-data storage in the cloud is largely handled by scalable key-value stores nowadays. Key-value stores, exposing a Put/Get API, allow only for key-based data access. In this chapter, I describe my HINDEX work that supports secondary index on write-optimized key-value stores with efficiency.

3.1 Introduction

In the age of cloud computing, various scalable systems emerge and prevail for big data storage and management. These scalable data stores, mostly called key-value stores, include Google's BigTable [53], Amazon's Dynamo [62], Facebook's Cassandra [91, 9], Apache HBase [14] among many others. They expose simple Put/Get API which allows only key-based data accesses, in the sense that when writing/reading data in the key-value stores, users are required to specify a data key as the parameter. While the key-based Put/Get API supports basic workloads, it falls short when it comes to more advanced web and database applications which require value-based data access. To gain wider application, it calls for value-based API support on the key-value stores.

On the other hand, many key-value stores deal with write-intensive big data. Typically, the workload against a key-value store is dominated by data writes (i.e. Put) rather than reads, and such kind of workloads is prevalent in modern web applications. For instance, in Web 2.0, social users not only read news but also contribute their own thinking and write news themselves. It is also the case in other emerging domains, such as large system monitoring and online financial trading. To optimize the write performance, many key-value

stores (e.g. HBase, Cassandra and BigTable) follow a log-structured merge design [99], in which on-disk data layout is organized to several sorted files and writes are optimized by an append-only design. We call these Log-structured Key-Value Stores as LSKV stores (whose distinctive features are described in § 3.2).

This work addresses the problem of *supporting a value-based API on the write-intensive data stored in* LSKV *store.* For value-based access, a secondary index is essential. In common practice, the secondary index is materialized as a regular table in the underlying LSKV store. In this situation, the index maintenance under a write-intensive workload is a challenge: On the one hand, the index maintenance needs to be lightweight in order for it to catch up with the high arrival rate of the incoming data writes; On the other hand, given multi-version data model in LSKV store, the index maintenance needs to find and delete old versions (in order to keep the index fresh and up-to-date), a task that includes Get operations and is very expensive in LSKV storesystems (explained in § 3.2).

In this chapter, we propose HINDEX, a middleware system that supports secondary index on top of an LSKV store. To address the index-maintenance challenge, we propose a performance-aware approach. The core idea is to decompose the index maintenance task to several sub-tasks, and only to execute the inexpensive ones synchronously while deferring the expensive ones. More specifically, given a data update, the index maintenance needs to perform two sub-tasks, that is, 1) to insert new data versions to the store and 2) to find and delete old versions. Sub-task 1) involves only Put operations, while sub-task 2), called index repair, requires a Get operation to find the old version. The insight here is that LSKV storeis write-optimized in the sense of fast Put and slow Get, which makes sub-task 1) lightweight and the index-repair sub-task 2) heavyweight. HINDEX's strategy to schedule the index maintenance is to synchronously execute sub-task 1) while deferring the expensive index-repair sub-task.

A core design choice regarding the deferred index repair is when the execution should

be deferred to. Our key observation is that a Get operation is much faster when it is executed after a compaction than before that. Here, a compaction is a native maintenance routine in LSKV store; it cleans up obsolete data and reorganizes the on-disk data layout. To verify the observation, we conducted a performance study on HBase 0.94.2. A preview of the experiment results is shown in Figure 8; the Get can achieve more than $7\times$ speedup in latency when executed after a compaction comparing to that before a compaction.[1] Based on this observation, we propose a novel design that *defers the index repair to the offline compaction process*. By coupling the index repair with the compaction it can save the index-repair overhead substantially.

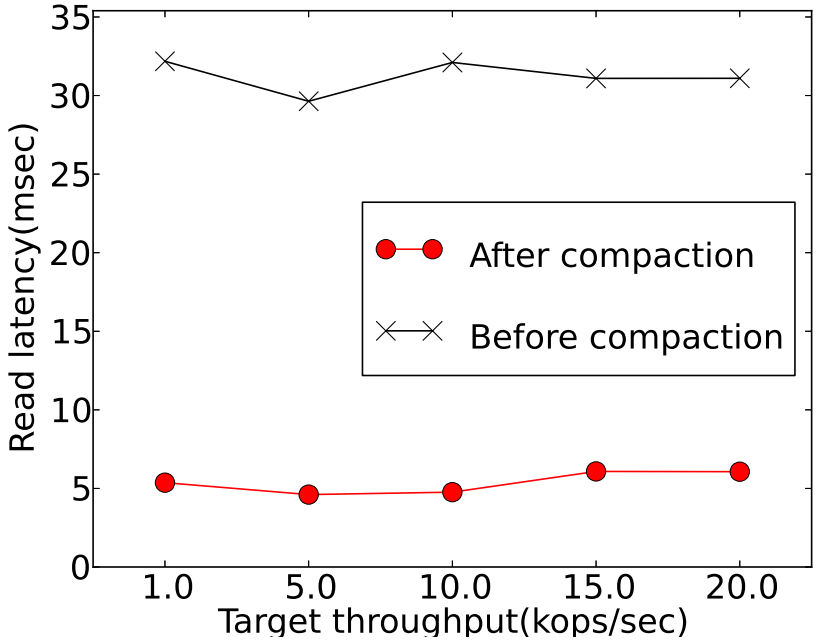


Figure 8: Read latency before/after compaction

The contributions of this chapter are summarized below.

- We coin the term LSKV store that abstracts various industrial strength big-data storage systems (including HBase, Cassandra, BigTable, HyperTable, etc). We propose HINDEX to extend the LSKV store's existing API by including a value-based access method.

[1] For details, please refer to the experiments in Section 3.6.2 and the explanation in Section 3.2.2.

- We make the index maintenance lightweight in HINDEX for write-intensive workloads. The core idea is performance awareness; it defers expensive operations while executing inexpensive ones synchronously in an LSKV storesystem. We also optimize the performance of deferred index maintenance by scheduling it with the compaction process.

- We analyze the fault-tolerance of HINDEX in terms of both online operations and offline index-repair process. The fault tolerance in HINDEX is achieved without sacrificing performance efficiency.

- HINDEX is designed to be generic on any LSKV store. While a generic implementation of HINDEX is realized in both HBase and Cassandra, we also demonstrate an HBase-specific implementation that optimizes performance without any internal code change in HBase (by using exposed system hooks).

3.2 Background: LSKV *store*

LSKV store, represented by BigTable [53]/HBase [14] and Cassandra [9], has the following two common and distinctive features.[2] Note that specific LSKV storesystems may differ in other aspects (e.g. HBase shards data by range partitioning while Cassandra is based on consistent hashing).

3.2.1 Key-Value Data Model

LSKV store employs a key-value data model with a Put/Get API. In the data model, a data object is identified by a unique key k and consists of a series of attributes in the format of key-value pairs; a value v is associated with multiple overwriting versions, each with a unique timestamp ts. To update and retrieve an object, LSKV store exposes a simple Put/Get API: Put(k, v, ts), Delete(k, ts) and Get$(k) \rightarrow \{\langle k, v, ts\rangle\}$. When calling these

[2]Note that other than LSKV store, there are key-value stores that are read optimized, such as PNUT [55].

API functions, the presence of a key k is required, which makes them key-based access methods.

3.2.2 LSM Tree-based Data Persistence

LSKV store adopts the design of LSM tree [99, 106] for its local data persistence. The core idea is to apply random data updates in append-only fashion, so that most random disk access can be translated to efficient, sequential disk writes. The read and write paths of an LSM tree are shown in Figure 9. After the LSM tree locally receives a Put request, the data is first buffered[3] in an in-memory area called *Memstore*[4], and at a later time is flushed to disk. This Flush process sorts all data in Memstore based on key, builds a key-based cluster index (called block index) in batch, and then persists both the sorted data and index to disk. Each Flush process generates an immutable file on disk, called *HFile*. As time goes by, multiple Flush executions can accumulate multiple HFiles on disk. On the data read path, an LSM tree process a Get request by sifting through existing HFiles on disk to retrieve multiple versions of the requested object. Even in the presence of existing in-memory index schemes (e.g., Bloom filters and the block index used in HBase), an LSM tree still has to access multiple files for a Get, because the append-only writes put multiple versions of the same object to different HFiles in a non-deterministic way. This design renders LSKV store to be a write-optimized system since a data write causes mostly in-memory operations, while a read has to randomly access the disk, causing disk seeks.

An LSKV store exposes a Compaction interface for system administrator to perform the periodical maintenance routine, usually in offline hours. A Compaction call triggers the compaction process, which merges multiple on-disk HFiles to one and performs data cleaning jobs to reclaim disk space from obsolete object versions. The Compaction consolidate the resource utilization in the LSKV store for further operation efficiency.

[3] For durability, LSM trees often have option for write-ahead logging (or WAL) before each write to buffer.
[4] In this chapter, we use the HBase terminology to describe an LSM tree.

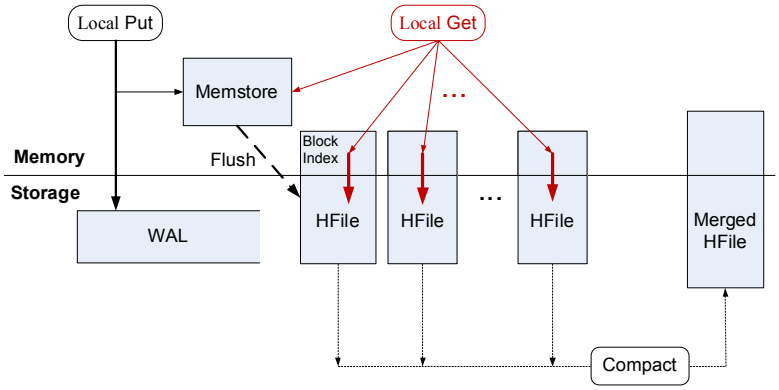


Figure 9: System architecture of an LSM tree: Black/red arrows denote the read/write path. Thick arrows represent disk access.

Fault tolerance An LSKV storeis fault-tolerant as long as it is on top of a reliable file system (e.g. HDFS). We can analyze the fault tolerance along the data write path in Figure 9: 1) For data in Memstore, it is fault tolerant because it is first persisted in the WAL; 2) Data in HFile is fault tolerant provided that HFile is stored reliably by HDFS. Upon machine failure, an LSKV storetypically does the following to recover: It relies on a global coordination service (e.g. Zookeeper) to detect the failure using a heart-beat mechanism, then retrieves the WAL data from the failed machine and re-assigns it to other live machines, which proceeds to replay the WAL data and restore the Memstore before it can be linked with previously persisted HFiles and serve the upcoming read and write requests.

3.2.3 Extension Interfaces in LSKV store

Following the trend of moving computation close to data, there is a recent body of work that enriches server-side functionality by adding extension interfaces to the key-value stores including HBase's CoProcessor [15], Cassandra's Plugins/Triggers [16, 17], Percolators' trigger functionality [101]. These extension interfaces typically expose event-based programming hooks and allow client applications to easily inject code (e.g., stored procedure) into store servers and associate that with their internal events. With the interfaces, one can extend the functionality of a key-value server without changing its internal code.

3.3 *The* HINDEX *Structure*

In this section we first present the system and data model in HINDEX, and then describe the materialization of HINDEX in the underlying LSKV store.

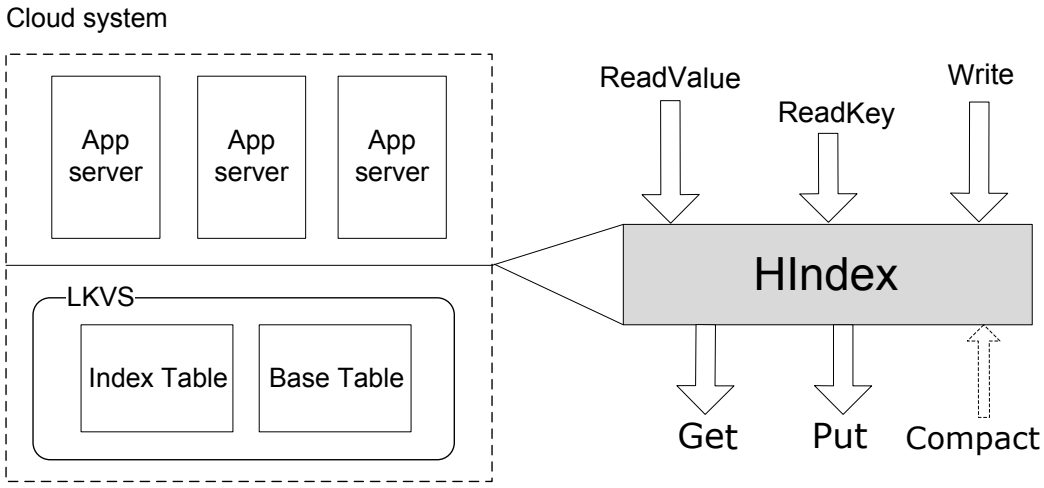


Figure 10: HINDEX architecture

3.3.1 System and Data Model

In a cloud environment, a server system is organized into a multi-tier architecture, consisting of application and storage tiers. The application tier processes queries and prepares the data formatting for the writes, while the storage tier is responsible for persisting the data. We consider the use of LSKV store in the storage tier, as shown in Figure 10. In this architecture, HINDEX is a middleware that resides between the application and storage tiers. To the application servers, it exposes both key-based and value-based API, as described below. The application servers are referred to as "client" (of HINDEX) in this chapter. To the underlying LSKV store, HINDEX translates the API invocations to the Put/Get operations.

- Write(k, v, ts): Given a row key k, it updates (or inserts) the value to be v with timestamp ts.
- ReadKey$(k, ts, m) \rightarrow \{\langle k, v', ts' \rangle\}_m$: Given a row key k, it returns the value versions before ts, that is, $ts' \leq ts$. HINDEX considers an m-*versioning* policy, which

allows client applications to indicate the number of versions deemed as fresh (by m). The method would return the latest m versions of the requested key.

- ReadValue$(v, ts, m) \rightarrow \{\langle k', v, ts' \rangle\}$: Given an indexed value v, it retrieves all the row keys k' whose values are v and which are valid under the m-versioning policy. That is, the result version ts' must be among the latest m versions of its key k' as of time ts.

The first two methods are similar to the existing key-based Put/Get interface (with different internal implementations), while the last one is for value-based data access. In the API design, we expose timestamp ts for the client applications to specify the consistency requirement. In practice, generating a unique timestamp, if necessary, can be done by existing timestamp oracles [128, 101].

HINDEX, being a generic index structure, can be adapted to various index and query types for different application purposes. While the rest of the chapter will primarily use the exact-match query to represent value-based accesses, it is straightforward to extend it to other query types. For example, the full-text search can be supported by having each row storing a document and having value v to be a keyword. In this case, ReadValue(v) returns a document that contains keyword v.

3.3.2 Index Materialization

The index data is materialized in a regular data table inside the underlying LSKV store. The index table is not directly managed by the client applications; instead, it is fully managed by our HINDEX middleware. In terms of structure, the index table is an inverted version of the base table; when the base table stores a (valid) key-value pair, say $\langle k, v \rangle$, the index table would store the reversed pair as $\langle v, k \rangle$. For different keys associated with the same value in the base table, HINDEX materializes them in the same row in the index table but as different versions. The versioning mechanism is disabled in the index table, and all obsolete index version is required to be deleted explicitly.

In the case of skewed data distribution, it could occur that certain index rows for common values are huge. It may result in a long list of query results by a ReadValue call. In this situation, we rely on a pagination mechanism to limit the number of key-value pairs in a ReadValue query result. Specifically, we allows an optional parameter p as in ReadValue$(v, ts, m, [p])$ which limits the number of ReadValue results to be under p. Currently, we assume it is the applications' responsibility to specify the value of p in the case of skewed data distribution, although automatically setting value p could be the future work.

3.4 Online HINDEX

This section describes the design of online HINDEX in terms of index maintenance, query evaluation, analysis of fault tolerance, and the implementation.

3.4.1 Put-Only Index Maintenance

Given a data update as a key-value pair $\langle k, v\rangle$, the index maintenance needs to include four tasks to keep both the index and base table up-to-date: 1) Deleting old versions associated with key k in the index table. This causes a Delete(v', k, ts') call, in which v' is the old version obsoleted by new version $\langle k, v\rangle$. Since old version v' is unknown from the original data update $\langle k, v\rangle$, it entails a Get operation to read the old version; 2) Reading the old version from the base table. This causes a call for Get$(k)\rightarrow$ $\langle k, v'\rangle$; 3) Inserting new version to the base table, which causes Put(k, v, ts); 4) Inserting new version to the index, which causes Put(v, k, ts).

A straightforward way to execute the index maintenance process is to *synchronously* execute all the four operations, which is essentially what the traditional update-in-place indexing technique does (which is also widely used in many cloud databases [41, 65]). However, this strategy causes performance problems when applied to the LSKV store case: Recall that a Get operation in LSKV store is slow, and by attaching expensive Get to the data write path, it could increase the write overhead and slow down the system throughput,

especially when the workload is dominated by data writes. To improve the online index maintenance efficiency, HINDEX employs a simple strategy to execute the Put-only [5] operations (i.e. operations 3) and 4)) synchronously and defer the execution of expensive index repair operations (i.e. operations 1) and 2)) to later time. Algorithm 3 illustrates the online Write algorithm. The two Put calls are annotated with the same timestamp ts. Here, we deliberately put the index update ahead of the base table update for the fault-tolerance concerns which will be discussed.

Table 4: Algorithms for online writes and reads

Algorithm 3 Write(key k, value v, timestamp ts)

```
1: index.Put(v, k, ts)
2: base.Put(k, v, ts)
```

Algorithm 4 ReadValue(value v, timestamp ts, versioning m)

```
1: {⟨k, ts'⟩} ← index.Get(v, ts)                          ▷ ts' ≤ ts
2: for ∀⟨k, ts'⟩ ∈ {⟨k, ts'⟩} do
3:     {⟨k, v', ts''⟩} ← ReadKey(k, ts, m)                ▷ ts'' is earlier than ts
4:     if ts' ∈ {⟨k, v', ts''⟩} then                      ▷ ts' is a fresh version regarding ts
5:         result_list.add({⟨k, v, ts'⟩})
6:     else
7:         if ts' > min {⟨k, v', ts''⟩} then
8:             index.Delete(v, k, ts')                    ▷ Cleanup dangling index data
9:         end if
10:    end if
11: end for
12: return result_list
```

3.4.2 Read-Query Evaluation

The Put-only index maintenance may lead to obsolete index data (e.g. $\langle v', k\rangle$ is present in the index table after $\langle k, v\rangle$ is written). This requires a ReadValue query to always check whether an index entry is fresh. Given an index data $\langle v', k\rangle$, the freshness check is done by checking with the base table, from which multiple value versions of key k are co-located at the same place and version freshness can be easily known. Algorithm 4 illustrates the

[5] Since Put is an append-only operation in LSKV store, we may use the term "Put-only" and "append-only" interchangeably in this chapter.

evaluation algorithm for ReadValue(v, ts, m): It first issues a Get call to the index table and reads the related index entries before timestamp ts. For each returned index entry, say ts', it needs to determine whether the entry is fresh under the m-versioning policy. To do so, the algorithm reads the base table by issuing a ReadKey query (which is a simple wrapper of a Get call to the base table), which returns all the latest m versions $\{ts''\}$ before timestamp ts. Depending on whether ts' show up in the list of $\{ts''\}$, the algorithm can then decide that it is fresh or obsolete. Only when the version is fresh, it is then added to the final result. If it is found that the index version ts' is not present in the base table, implying the occurrence of a failure, it issues a Delete call to remove the dangling index data.

3.4.3 Fault Tolerance

In a cloud environment where machines fail, it is possible that a Write can fail with only one Put completed. To deal with failure, our API has the following semantics.

- A Write is considered to succeed only when both Put operations complete, and a read (either ReadKey or ReadValue) will not return any data that is not written successfully.

Under this semantics, HINDEX can achieve consistent data reads and writes with efficiency. We consider the scenario where the machine issuing a Write call fails. It is a trivial case when the failure occurs either before or after the Write invocation; in this case, nothing inconsistent will be left in the LSKV storeand this can be guaranteed by the fault-tolerance and atomicity property of the underlying LSM tree (see Section 3.2.2). Thus, what is interesting is the case that failure happens between index.Put(v, k, t) and base.Put(k, v, t) in the write path. This case can lead to dangling index rows without corresponding data stored in the base table. This (inconsistent) situation can affect the reads under three circumstances: 1) ReadValue(v), 2) ReadValue(v') where v' is the version right before v and 3) ReadKey(k). For case 1), Algorithm 4 is able to discover the dangling index data (as in Line 7) and would correctly neglect such data to comply with our API semantics. It

actually issues an index.Delete($\langle v, k \rangle$) to remove the dangling data from being considered by future ReadValue(v) invocations. For case 2), Algorithm 4 will not find any version that overwrites v' in the base table and would return $\langle k, v' \rangle$. For case 3), ReadKey returns $\langle k, v' \rangle$. In all cases, our API semantics holds. It is fairly easy to extend this analysis to multiple failed Write's.

3.4.4 Implementation of Online HINDEX

Online HINDEX can be implemented on LSKV stores by two approaches. The first approach is to implement it in the client library. That is, the client directly coordinates all the function calls (e.g. Write and ReadKey) as in the Write and ReadValue algorithms; This is possible since all the operations in these algorithms are based on the generic Put/Get interface. The second approach is server-side implementation. In this case, the index and base table servers play the role of coordinators to execute the read and write algorithms. In particular, the Write is rewritten to a base-table Put by the HINDEX client library. When the base-table Put gets executed in the base table, it also triggers the execution of the index-table Put. Likewise, the ReadValue is rewritten to an index-table Get call, upon the completion of which the index table triggers the execution of the base-table Get, if needed. The server-side implementation favors the case where application servers and storage servers are located in different clusters and the cross-boundary inter-cluster communications are more expensive than the intra-cluster communications. The server-side implementation can be done by directly modifying the code of an LSKV store system, or as in our implemented prototype, by adding server extensions based on the extension interface of LSKV stores.

3.5 Offline HINDEX*: Batched Index Repair*

In HINDEX, the index repair process eliminates the obsolete index entries and can keep the index fresh and up-to-date. This section describes the design and implementation of an offline and batched repair mechanism in HINDEX.

3.5.1 Computation Model and Algorithm

To repair the index table, it is essential to find the obsolete data versions. A data version, say $\langle v', k, ts' \rangle$, is considered to be obsolete when either of the following two conditions is met.

1. There are at least m newer key-value versions of key k that exist in the system.

2. There is at least one newer Delete tombstone[6] of key k that exists in the system.

The process to find the obsolete versions, called index garbage collection, is realized by scanning the base table. Because the base table has the data sorted in the key order, which helps verify the above two conditions. Algorithm 5 illustrates the batched garbage collection algorithm on a data stream that is output from the table scan; the data stream is assumed to have key-value pairs ordered first by key and then by timestamp. The algorithm maintains a queue of size m and emits the version only when it is older than at least m versions of the same key k in the queue and it is repaired by previous repair processes (will be discussed in next paragraph). In the algorithm, it also considers the condition regarding a Delete tombstone; it will emit all the versions before the Delete tombstone marker. Note that our algorithm requires a small memory footprint (i.e. the queue of size m).

Our offline index repair process runs periodically (e.g. on a daily or weekly basis). To avoid duplicated work between multiple rounds of repairs, we require that each run of an index repair process is marked with a timestamp, so that the versions of interest to this run are those with timestamps falling in between the timestamp of this run and that of previous run (i.e. ts_{Last}). Any version that is older than ts_{Last} was repaired before by previous index-repair processes and is not considered in the current run.

[6] In an LSKV store, a Delete operation appends a tombstone marker in the store without physically deleting the data.

Algorithm 5 BatchedGC(Table-scan stream s, versioning m, ts_{Last})

```
1: for ∀⟨k, v, ts⟩ ∈ s do                    ▷ Stream data sorted by key and time (in descending order)
2:     if k_Current == k then
3:         if queue.size() < m then
4:             queue.enqueueToHead(⟨k, v, ts⟩)
5:         else if queue.size == m then
6:             queue.enqueueToHead(⟨k, v, ts⟩)
7:             ⟨k, v′, ts′⟩ ← queue.dequeueFromTail()    ▷ All pairs in the queue are of the same
   key k_Current = k
8:             if ts′ ≥ ts_Last then                      ▷ ts′ is no older than ts_Last
9:                 emit(⟨k, v′, ts′⟩)
10:            end if
11:        end if
12:    else
13:        loop queue.size()> 0                           ▷ Clear the queue
14:            ⟨k, v′, ts′⟩ ← queue.dequeueFromHead()
15:            if ⟨k, v′, ts′⟩ is a Delete tombstone then
16:                emit(queue.dequeueAll())
17:            end if
18:        end loop
19:        k_Current ← k
20:        queue.enqueueToHead(⟨k, v, ts⟩)
21:    end if
22: end for
```

3.5.2 Compaction-Aware System Design

To materialize the table scan in the presence of an offline compaction process, one can generally have three design options, that is, to run the index repair 1) before the compaction, 2) after the compaction or 3) coupled inside the compaction. In HINDEX, we adopt the last two options (i.e. to couple the index repair either after or within the compaction). Recall that in an LSKV store, the read performance is significantly improved after the compaction. The rationale of such design choice is that the table scan, being essentially a batch of reads, has its performance dependent on the compaction: Without the compaction, there would be a number of on-disk files and a key-ordered scan would essentially become a batch of random reads that make the disk heads swing between the multiple on-disk files.

Overall, the offline HINDEX runs in three stages; as illustrated in Figure 11, it runs the offline compaction, garbage collection and index garbage deletion. After a Compaction call is issued, the system runs the compaction routine, which then triggers the execution of index garbage collection and deletion (for the index repair). Specifically, the garbage collection emit the obsolete data versions to the next-stage garbage deletion process. The index garbage deletion issues a batch of deletion requests to the distributed index table. In the following, we describe our subsystems for each stage and discuss the design options.

3.5.2.1 The Garbage Collection

We present two system designs for garbage collection, including an isolated design that puts the garbage collection right after the compaction process, and a pipelined design that couples the garbage collection inside the compaction process.

An isolated design The garbage collection subsystem is materialized as an isolated component that runs after the previous compaction completes. As portrayed in Figure 11, the system monitors the number of sorted data files in the local machine. When an offline compaction process finishes, it reduces the number of on-disk files to one, upon which the

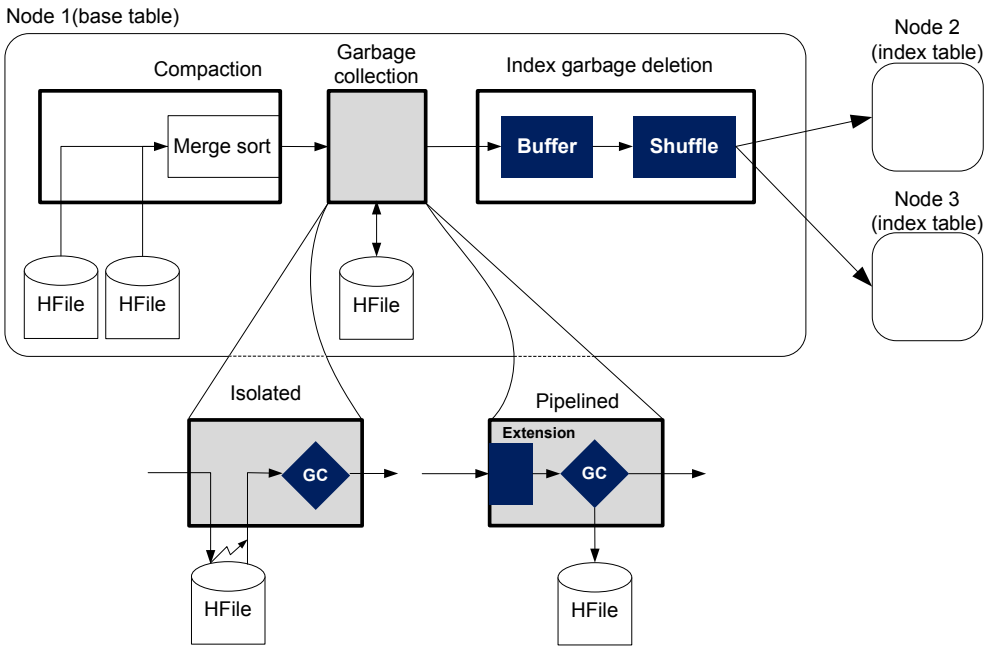


Figure 11: Compaction-aware index repair

monitor component triggers the garbage collection process. In this case, the garbage collection reloads the newly generated file to memory (and the file system cache may still be hot), scan it, and run Algorithm 5 to collect the obsolete data versions.

A pipelined design Alternatively, the garbage collection subsystem can be implemented by pipelining the compaction's output stream directly to the garbage collection service. To be specific, as shown in Figure 11, the pipelined garbage collection intercepts the output stream from the compaction while the data is still in memory; the realization of interception will be described below. Then it runs the garbage collection computation in Algorithm 5; if the data versions are found to be obsolete, they are emitted but still being persisted (for fault-tolerance concerns). Comparing the isolated design, the pipelined design saves disk accesses, since the data stream is directly pipelined in memory without being reloaded from disk.

In terms of generosity, while the isolated design can be implemented on any key-value

stores as it requires nothing more than a generic file abstraction, the pipelined implementation may benefit from certain system hooks (e.g. the ones exposed by HBase). We implemented the isolated system on both HBase and Cassandra, and implemented the pipelined system on HBase.

Implementation of Pipelined Repair The compaction's output stream can be intercepted by various approaches. The most straightforward and generalized approach is to directly modify the internal code of an LSKV store. Another approach is to rely on the extension interface widely available in existing LSKV stores (described in Section 3.2.3) which allows for an add-on and easier implementation. In particular, our HBase-based prototype implements the pipelined design using extension-based implementation; We register a CoProcessor callback function to hook the garbage collection code inside the compaction. By this way, our implementation requires no internal code change of HBase, and can even be deployed lively onto a running HBase cluster.

3.5.2.2 The Index Garbage Deletion

For each key-value pair emitted from the garbage collection, it enters the garbage deletion stage; the data is first buffered in memory and later shuffled before being sent out by a Delete call to the remote index table. The shuffle process sorts and clusters the data pairs based on the value. By this way, the reversed value-key pairs with the same destination can be packed into a single serializable object in a RPC call, thus network utilization can be improved. In the design of the garbage deletion subsystem, we expose a tunable knob to configure the maximal buffer size; The bigger the buffer is, the more bandwidth efficiency it can achieve at the expense of more memory overhead.

3.5.3 Fault Tolerance

To design our index-repair protocol with fault tolerance, we enforce the following property:

- Given a compaction and index-repair process with ts_{Last}, it does not physically delete any data of interest (i.e. with timestamp between now and ts_{Last}).

Note that for any data before ts_{Last}, physical deletion is enabled.

Based on this property and idempotency of underlying LSKV store (i.e. there is no additional effect if a Put is called more than once with the same parameters), we can easily guarantee the fault tolerance of the index-repair process. The logic is following: Given a failed run of index repair, we can simply ignore its partial results and keep the old ts_{Last}. Upon the next run of index repair, the above property guarantees that all data versions from ts_{Last} to the present are still there in the system and the current run, if it succeeds, will eventually repair the index table correctly. Note that since the previously repaired data is not deleted, it may cause some duplicated operations which however do not affect the correctness due to idempotency.

3.6 Experiments

This section describes our experimental evaluation of HINDEX. We first did experiments to study the performance characteristics of HBase, a representative LSKV store, and then to study HINDEX's performance under various micro-benchmarks and a synthetic benchmark. Before all of these, we describe our experiment system setup.

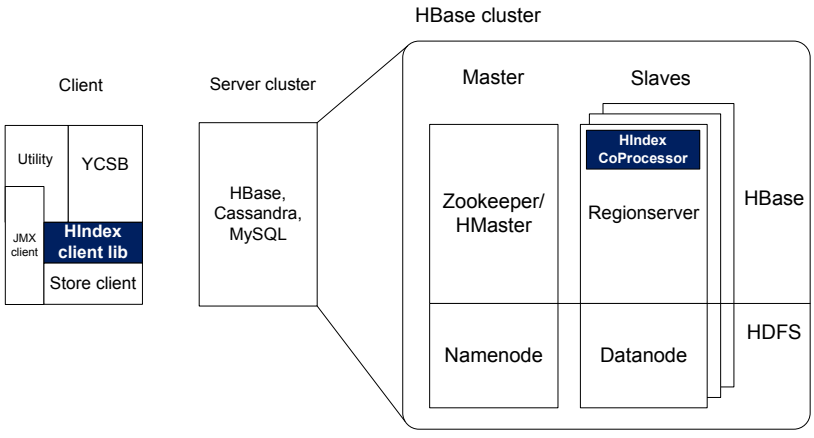


Figure 12: Experiment platform and HINDEX deployment

3.6.1 Experiment System Setup

The experiment system, as illustrated in Figure 12, is organized in a client/server architecture. In the experiment, we use one client node and a 19-node server cluster, consisting of a master and 18 slaves. The client connects to both the master and the slaves. We set up the experiment system by using Emulab [18, 126]; all the experiment nodes in Emulab are homogeneous in the sense that each machine is equipped with the same 2.4 GHz 64-bit Quad Core Xeon processor and a 12 GB RAM. In terms of the software stack, the server cluster uses both Hadoop HBase and HDFS [13]. The HBase and HDFS clusters are co-hosted on the same set of nodes, as shown in Figure 12. Unless otherwise specified, we use the default configuration in the out-of-box HBase. The client side is based on the YCSB framework [56], an industry-standard benchmark tool for evaluating the key-value store performance. The original YCSB framework generates only key-based queries, and for testing our new API, we extended the YCSB to generate value-based queries. We use the modified YCSB framework to drive workload into the server cluster and measure the query performance. In addition, we collect the system profiling metrics (e.g. number of disk reads) through a JMX (Java management extension) client. For each run of experiments, we clean the local file system cache.

HINDEX **prototype deployment** We have implemented an HINDEX prototype in Java and on top of HBase 0.94.2. The HINDEX prototype is deployed to our experiment platform in two components; as shown by dark rectangular in Figure 12, the HINDEX middleware has a client-side library for the online operations and a server-side component for the offline index repair. In particular, based on system hooks in HBase, the prototype implements both the isolated garbage collection and pipelined garbage collection in the server component.

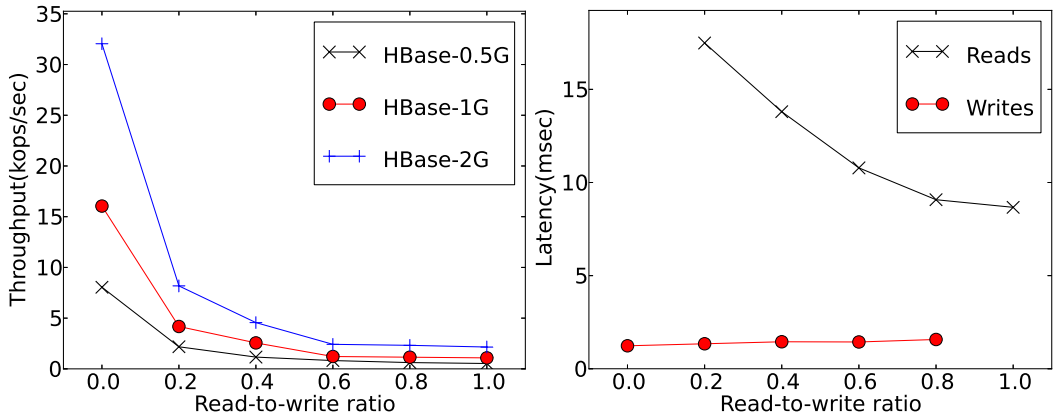
Dataset Our raw dataset consists of $1000,000,000$ key-value pairs, generated by YCSB using its default parameters that simulates the production use of key-value stores inside

Yahoo!. In this dataset, data keys are generated in a Zipf distribution and are potentially duplicated, resulting in $20,635,449$ distinct keys. The data values are indexed. The raw dataset is pre-materialized to a set of data files, which are then loaded to the system for each experiment run. For query evaluation, we use $1,000,000$ key-value queries, be it either Write, ReadValue or ReadKey. The query keys are randomly chosen from the same raw dataset, either from the data keys or values.

3.6.2 Performance Study of HBase

Read-write performance This set of experiments evaluates the read-write performance in the out-of-box HBase to verify that HBase is aptly used in a write-intensive workload. In the experiment, we set the target throughput high enough to saturate the system. We configure the JVM (on which HBase runs) with different heap sizes. We varied the read-to-write ratio [7] in the workload, and report the maximal sustained throughput in Figure 13a, as well as the latency in Figures 13b. In Figure 13a, as the workload becomes more read intensive, the maximal sustained throughput of HBase decreases, exponentially. For different JVM memory sizes, HBase exhibits the similar behavior. This result shows that HBase is not omnipotent but particularly optimized for write-intensive workloads. Figure 13b depicts the latency respectively for reads and writes (i.e. Get and Put) in HBase. It can be seen that the reads are much slower than writes, by an order of magnitudes. This result matches the system model of LSKV store in which reads need to check more than one places on disk and the writes are append-only and fast. In the figure, as the workload becomes more read intensive, the read latency decreases. Because with read-intensive workload, there are fewer writes and thus fewer data versions in the system for a read to check, resulting in faster reads.

[7] In the chapter, the read-to-write ratio refers to the percentage of reads in a read-write workload.



(a) Maximal sustained throughput (b) Latency

Figure 13: HBase performance under different read ratios

Read Performance and HFiles This experiment evaluates HBase's read performance under varying number of HFiles. In the experiment, we start with preloading the dataset into the HBase cluster, which results in averagely 11 HFiles in each region sever as shown in Figure 8. During the experiment, we issued every 5 million Put's to the HBase cluster and measure the read latency by issuing a number of Get's then. In this process, we have configured the HBase to disable its automatic compaction and region split and during the read performance evaluation, we disallow any Put operations, so that the number of HFiles would stay constant at that time. We measure the number of HFiles between each Put stage. As shown in Figure 8, the average number of HFiles increases from 11 to 27.5 in the experiment. In the end, we manually issued an offline Compaction call across all the regions in the cluster, which should leave all the regions with a single HFile. Then the read latency is measured again. We report the changes of read latency in this process in Figure 14 with the red line. As can be seen, the line of read latency basically matches with that of number of HFiles; As there are more HFiles present in HBase, the read latency also becomes bigger. After the compaction which merge all HFiles into one, the read latency also drops greatly. The experiment shows the strong dependency between the Get latency and the number of HFiles in HBase. In particular, for the state with 27.5 HFiles and the final state with 1 HFiles, we have shown the latency difference previously in Figure 8.

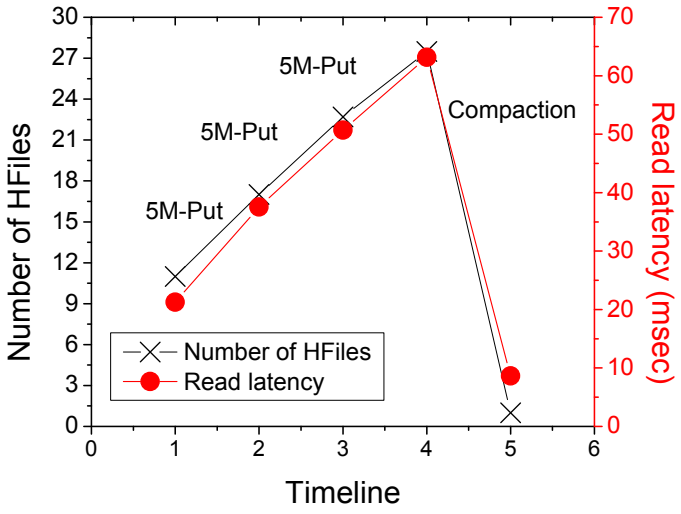
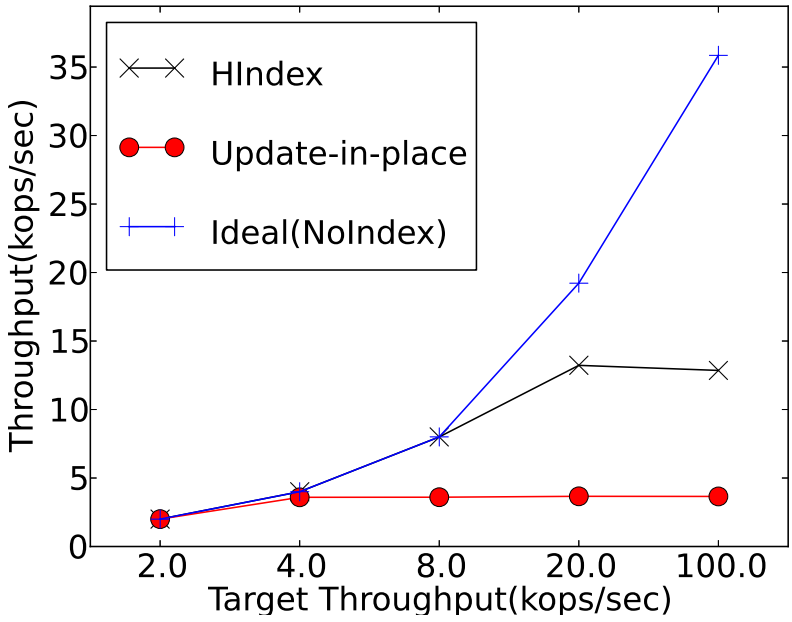


Figure 14: Read latency with varying number of HFiles

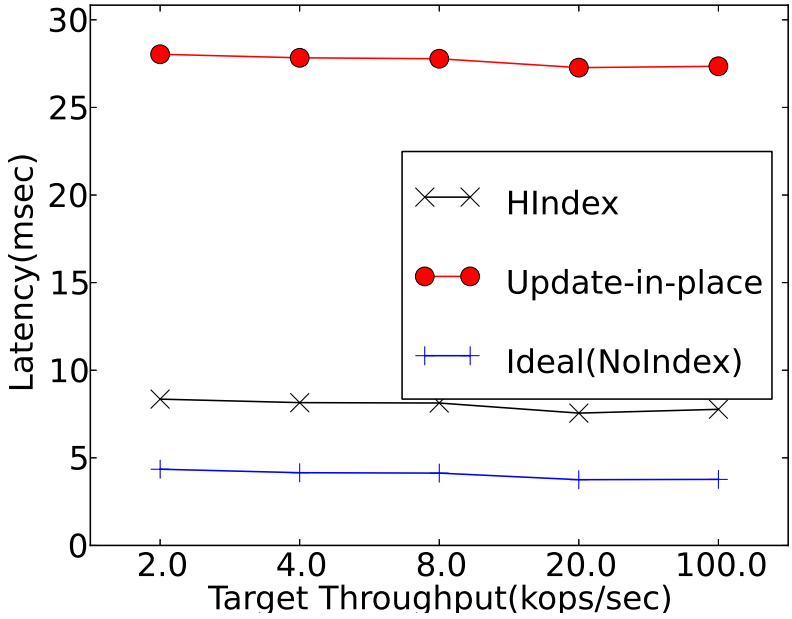
3.6.3 HINDEX Performance

Online write performance This experiment evaluates HINDEX performance under the write-only workloads. We drive the data writes from the client into the HBase server cluster. We compare HINDEX with the update-in-place indexing approach described in Section 3.4.1. We also consider the ideal case where there is no index structure to maintain. The results of sustained throughput are reported in Figure 15. As the target throughput increases, the update-in-place indexing approach hits the saturation point much earlier than HINDEX. While HINDEX can achieve a maximal throughput at about 14 thousand operations (kops) per second, the update-in-place indexing approach can only sustain at most 4 kops per second. Note that the ideal case without indexing can achieve higher throughput but can not serve the value-based queries. This result leads to a $3\times$ performance speedup of HINDEX. In terms of the latency, Figure 15b illustrates that HINDEX constantly outperforms the update-in-place approach under varied throughput.

Online read-write performance In this experiment, we evaluate HINDEX's performance in the workload that varies from read-intensive workloads to write-intensive ones. We



(a) Maximal throughput



(b) Latency

Figure 15: Index write performance

compare HINDEX on top of HBase against two alternative architectures: the B-tree index in MySQL and the update-in-place indexing on HBase. For fair comparison, we use the same dataset in both HBase and MySQL, and drive the same workload there. MySQL is

accessible to YCSB through a JDBC driver implemented by us, in which we reduce as much as possible the overhead spent in the JDBC layer. The results are shown in Figure 16. With varying read-to-write ratios, HINDEX on HBase is clearly optimized toward write-intensive workload, as can be seen in Figure 16a. On a typical write-intensive setting with 0.1 read-to-write ratio, HINDEX on HBase outperforms the update-in-place index on HBase by a $2.5\times$ or more speedup, and the BTree index in MySQL by $10\times$. When the workload becomes more read-intensive, HINDEX may become less advantageous. By contrast, the update-in-place approach is more read-optimized and the BTree index in MySQL is inefficient, regardless of workloads. This may be due to that MySQL uses locking intensively for full transaction support, an overkill to our targeted use case. In terms of latency, the HINDEX on HBase has the lowest write latency at the expenses of relatively high read latency due to the extra reads to the base table. By contrast, the update-in-place index has the highest write latency (due to the reads of obsolete versions in the base table) and a low read latency (due to that it only needs to read the index table). Note that in our experiments, we use more write-intensive values for read-to-write ratios (e.g. more ticks in interval $[0, 0.5)$ than in $[0.5, 1.0]$).

Offline index repair performance This experiment evaluates the performance of offline index repair with compaction. We mainly focus on the approach of compaction-triggered repair in the offline HINDEX; in the experiment we tested two implementations, with isolated garbage collection and pipelined garbage collection. For comparison, we consider a baseline approach that runs the batch index repair before (rather than after) the compaction (i.e. design option 1) in Section 3.5.2). We also test the ideal case in which an offline compaction runs without any repair operations. During the experiment, we tested two datasets: a single-versioned dataset that is populated with only data insertions so that each key-value pair has one version, and a multi-versioned dataset populated by both data insertions and updates which results in averagely 3 versions for each data value. While

the multi-versioned data is used to evaluate both garbage collection and deletion during the index repair, the single-versioned dataset is mainly used to evaluate the garbage collection, since there are no obsoleted versions to delete. In the experiment, we have configured the buffer size to be big enough to accommodate all obsolete data in memory. [8] We issued an offline Compaction call in each experiment, which automatically triggers the batch index repair process. Till the end, we collect the system profiling information. In particular, we collect two metrics, the execution time and the total number of disk block reads. Both metrics are emitted by the HBase's native profiling subsystem, and we implemented a JMX client to capture those values.

We run the experiment three times, and report the average results in Figure 17. The execution time is reported in Figure 17a. In general the execution time with multi-versioned dataset is much longer than that with the single-versioned dataset, because of the extra need for the index deletion. Among the four approaches, the baseline is the most costly because it loads the data twice and from the not-yet-merged small data files, implying that disk reads are mostly random accesses. The ideal case incurs the shortest time, as expected. Between the two HINDEX designs, the pipelined garbage collection requires shorter time because it only needs to load the on-disk data once. To understand the performance difference, it is interesting to look at the disk read numbers, as shown in Figure 17b. We only show the results with the single-versioned dataset, because disk reads only occur in the garbage collection. The baseline approach incurs a similar number of disk reads to the isolated design, because both approaches load the data twice from the disk. Note that the disk reads in the baseline approach are most random access while at least half of disk access in the isolated HINDEX should be sequential; this leads to differences in their execution time. In Figure 17b, the ideal case has a similar cost to the pipelined design, because both approaches load on-disk data once. From the single-versioned results in Figure 17a,

[8] We try to set up our experiment to be more bounded by local disk accesses than by the network communications, so that the offline index repair process is dominated by the garbage collection process than the deletion process.

it can be seen that their execution time is also very close to each other, due to that the extra garbage collection caused by the HINDEX approach is very lightweight and incurs few in-memory computations.

Table 5: Overhead under Put and Compaction operations

Name	Exec. time (sec)		Number of disk reads	
HINDEX	**1553.158**		60699	
Update-in-place index	4619.456		313662	
Name	Online	Offline	Online	Offline
HINDEX	**1093.832**	459.326	0	60699
Update-in-place index	4340.277	**279.179**	252964	60698

Mixed online and offline operations In this experiment, we compare HINDEX and the update-in-place indexing approach as a whole package. In other words, we consider the online and offline operations together. Because the update-in-place approach already repairs the index in the online phase, there is no need to perform index repair in the offline time. For fair comparison, we run the offline compaction (without any repair actions) for the update-in-place index. In the experiment, the online workload contains a series of writes and the offline workload simply issues a Compaction call and if any, the batch index repair. For simplicity, we here only report the results of pipelined HINDEX. We report the execution time and the number of disk reads. The results are presented in Table 5. In general, HINDEX incurs much shorter execution time and fewer disk reads than the update-in-place approach. For example, the execution time of HINDEX (in bold text in the table) is one third of that of the update-in-place approach. We breaks down the results to the online costs and offline costs, as in the bottom half of the table, which clearly shows the advantage of having the index repair deferred to the offline phase in HINDEX. Although the update-in-place index wins slightly in terms of the offline compaction (see the bold text "279.179" compared to "459.326" in the table), HINDEX wins big in the online computation phase

(see bold text "1093.832" compared to "4340.277" in the table). It leads to overall performance gain of HINDEX. In terms of disk reads, it is noteworthy that HINDEX incurs zero costs in the online phase.

3.7 Related Work

Secondary indexes in key-value stores Recently, a large body of academic work [41, 34, 65] and industrial projects [73, 28, 19, 43, 12, 108, 20] emerge to build secondary indexes as middleware on scalable key-value store systems. Those systems can be largely categorized by their design choices in terms of: 1) whether the index is local or global, 2) how the index is maintained, and 3) the system implementation. Regarding choice 2), the index can be maintained synchronously, asynchronously, or in a hybrid way. Synchronous index maintenance indicates real-time query result availability at the expense of extra index update overhead. Asynchronous index maintenance means the whole index updates are deferred. The hybrid approach is essential the append-only design (as in HINDEX) in which only the expensive part of index maintenance is deferred. In terms of implementation, the index middleware can be in the client or server side, depending on the preference on system generosity or performance. We summarize these key-value store indexes in Table 6.

In particular, Megastore [43] is Google's effort to support the cloud-scale database on the BigTable storage [53]. Megastore supports secondary indexes at two levels, namely the local index and global index. The local index considers the data from an "entity group" of machines that are close by. When the entity group is small, the local index is maintained synchronously at low overhead. The global index which spans cross multiple groups is maintained asynchronously in a lazy manner. F1 [108], built on top of Spanner [57], supports global indexing in fully consistent and transactional fashion; it applies 2PC with reasonably low costs. PIQL [41] supports a scale-independent subset of SQL queries including the value-based selection queries. The index maintenance in PIQL is based on the update-in-place approach. In addition, to support analytical queries, prior work [34]

Table 6: Key-value indexing systems (– means uncertain and * means HINDEX is implemented on HBase and Cassandra.)

References	Local/Global	Index Mntn	Impl.
Phoenix [28]	Global	Hybrid	HBase-Server
Huawei's Index [19]	Local	Sync	HBase-Server
Cassandra Index [12]	Local	–	Cassandra-Server
HyperTable Idx [20]	Global	–	HyperTable-Server
Megastore [43]	Local/global	Sync/Async	BigTable-Client
F1 [108]	Global	Sync	Spanner[57]-Client
PIQL [41]	Global	Sync	SCADR [42]-Client
HINDEX	Global	Hybrid	General*-Client/Server

maintains a global materialized view asynchronously and a local materialized view synchronously on top of PNUTS [55]. The local view is mainly for processing aggregation and ad-hoc queries, while the global view helps evaluate the selection queries. HyperDex [65] builds a key-value store from scratch to support a native ReadValue operation in the presence of multi-dimensional data.

While existing literature considers the append-only index maintenance (e.g. Phoenix [28]), it does not address the problem of scheduling expensive index-repair operations, which may lead to an eventually inconsistent index and cause unnecessary cross-table check for query processing. By contrast, the HINDEX design is aware of the asymmetric performance characteristic in an LSKV store and optimizes the execution of index repairs accordingly.

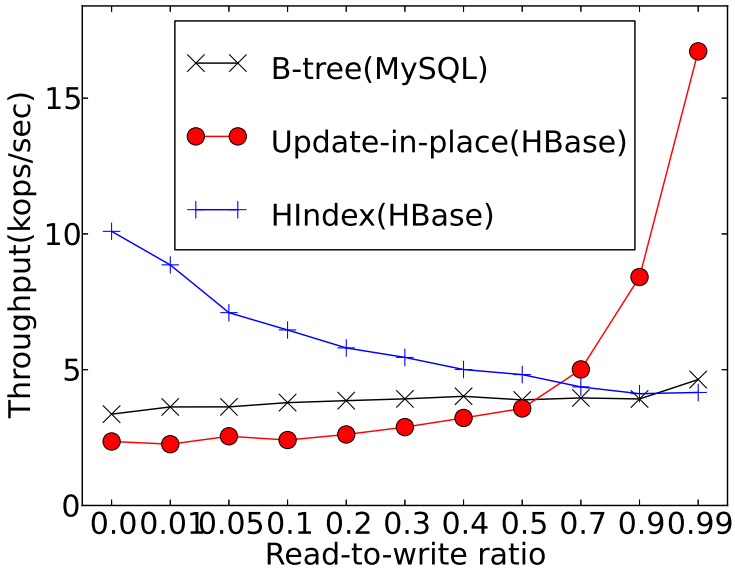
Log-structured systems Log-structured systems have been studied for more than two decades in the system community. The existing work largely falls under two categories, the unsorted LFS-like systems [105] and sorted LSM tree-like systems. While the former maintains a global log file in which data is appended purely by the write time, the latter organizes the data layout to a number of spill files, in each of which data is sorted based on the key. Log-structured systems generally rely on a garbage collection mechanism to reclaim disk space and/or re-organize the data layout. In particular, several heuristic-based garbage collection policies [49, 107] are proposed and adopted in LFS systems.

Recently, due to the burst of write-intensive workloads, log-structured design has been

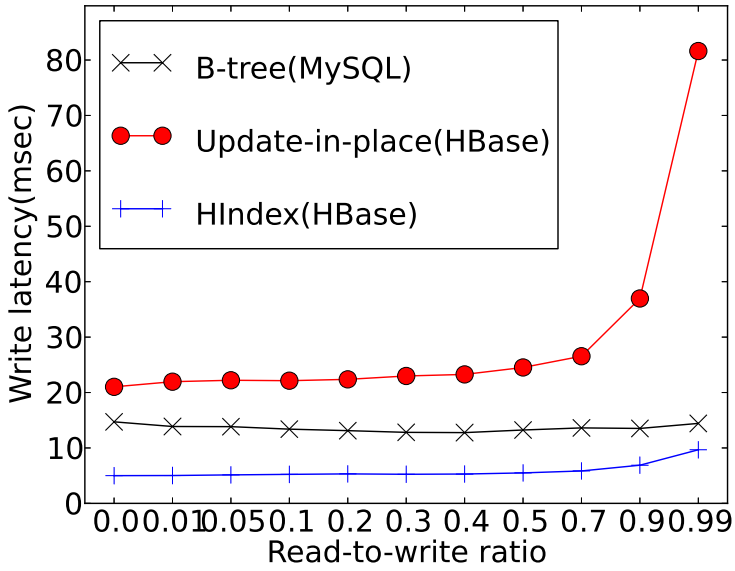
explored in the context of big data systems in the cloud. In addition to various LSKV store systems, bLSM [106] aims at improving the read performance on log-structured stores; the idea is to decompose the cumbersome compaction process so that it can be run at fine granularity with costs being piggybacked with other concurrently running operations. Several key-value stores adopt the unsorted LFS design. Based on a farm of Flash/SSD storage nodes, FAWN [36] avoids the costly in-place writes on SSD by a sequential log and maintains an in-memory index that is updated in place and can speed up the random reads. RAMCloud [100] builds an in-memory data management system in Cloud with a focus of efficient data recovery.

3.8 Conclusion

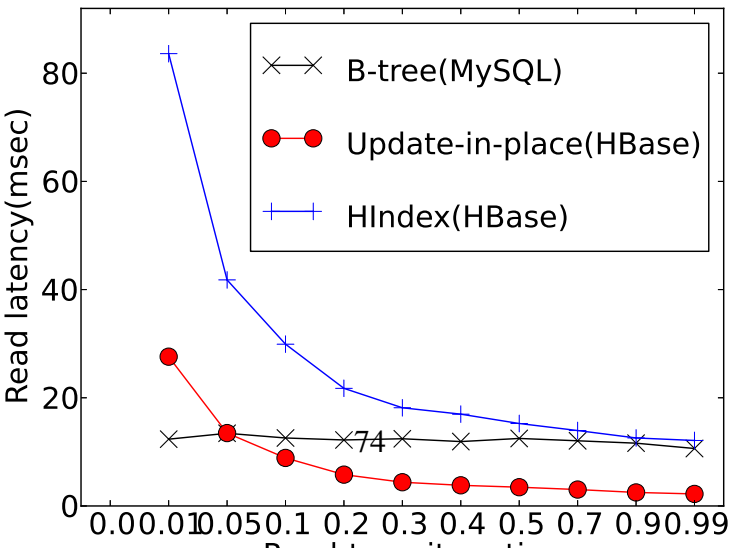
This chapter describes HINDEX, a lightweight real-time indexing framework for generic log-structured key-value stores. The core design in HINDEX is to perform the *append-only* online indexing and compaction-triggered offline indexing. By this way, the online index update does not need to look into historic data for in-place updates, but rather appends a new version, which substantially facilitates the execution. To fix the obsolete index entries, HINDEX performs an offline batched index repair process. By coupling with the native compaction routine in an LSKV store, the batch index repair achieves significant performance improvement by incurring no extra disk accesses. We implemented an HINDEX prototype based on HBase and demonstrate the performance gain by conducting experiments in real-world system setup.

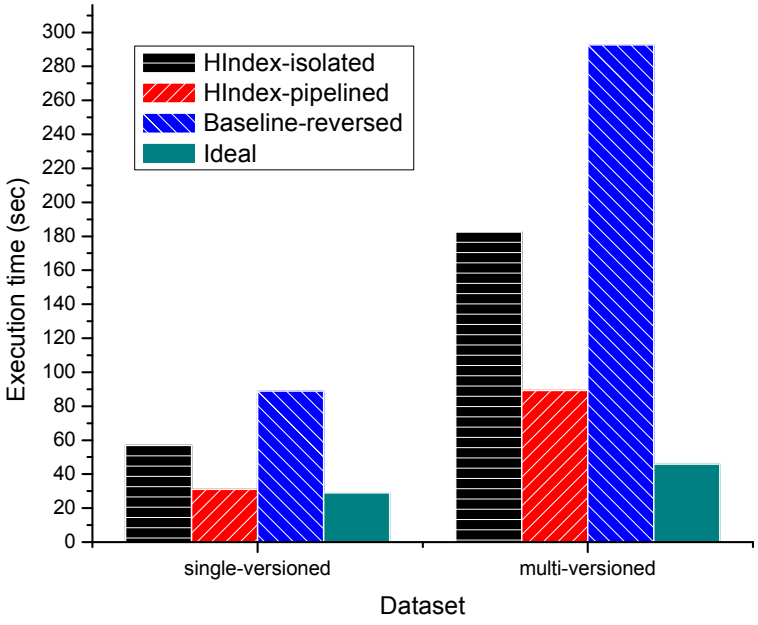


(a) Maximal throughput

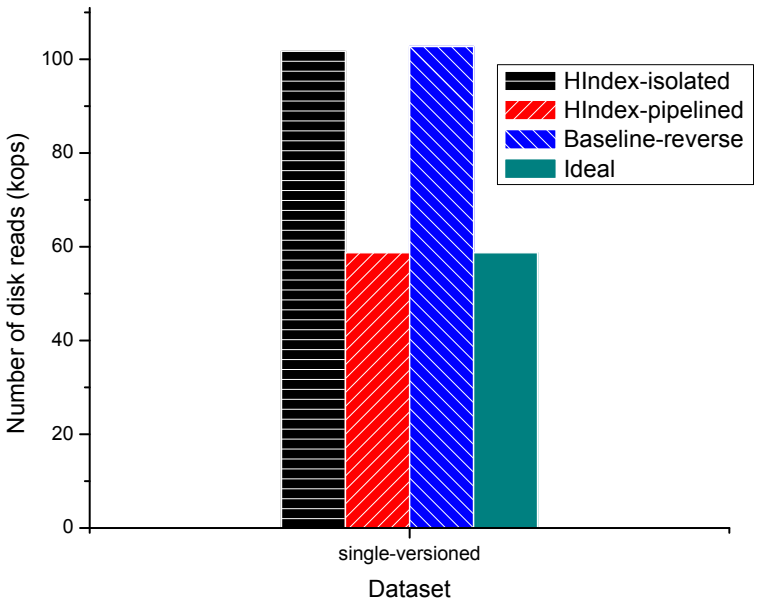


(b) Write latency





(a) Execution time



(b) Number of disk reads

Figure 17: Performance of offline index repair

CHAPTER IV

AUTOPIPELINING: AUTOMATIC PERFORMANCE OPTIMIZATION OF STREAMING APPLICATIONS ON MULTI-CORE MACHINES

A system in the cloud typically deals with large number of machines, managing which manually would be very labor-intensive work. Automating administrative tasks in the cloud, particularly for performance optimization, would be critically important. In this chapter, I describe my work on automatic pipelining for streaming applications on multi-core machines in the cloud.

4.1 Introduction

With the recent explosion in the amount of data available as live feeds, *stream computing* has found wide application in areas ranging from telecommunications to health-care to cyber-security. Stream processing applications implement data-in-motion analytics to ingest high-rate data sources, process them on-the-fly, and generate live results in a timely manner. Stream computing middleware provides an execution substrate and runtime system for stream processing applications. In recent years, many such systems have been developed in academia [40, 33, 82], as well as in industry [111, 69, 129].

For the last decade, we have witnessed the proliferation of multi-core processors, fueled by diminishing gains in processor performance from increasing operating frequencies. Multi-core processors pose a major challenge to software development, as taking advantage of them often requires fundamental changes to how application code is structured. Examples include employing thread-level primitives or relying on higher-level abstractions that

have been the focus of much research and development [26, 2, 104, 52, 84, 54]. The high-throughput processing requirement of stream processing applications makes them ideal for taking advantage of multi-core processors. However, it is a challenge to keep the simple and elegant data flow programming model of stream computing, while best utilizing the multiple cores available in today's processors.

Stream processing applications are represented as data flow graphs, consisting of reusable operators connected to each other via stream connections attached to operator *ports*. This is a programming model that is declarative at the flow manipulation level and imperative at the flow composition level [71]. The data flow graph representation of stream processing applications contains a rich set of parallelization opportunities. For instance, *pipeline parallelism* is abundant in stream processing applications. While one operator is processing a tuple, an upstream operator can process the next tuple concurrently. Many data flow graphs contain bushy segments that process the same set of tuples, and which can be executed in parallel. This is an example of *task parallelism*. It is noteworthy that both forms of parallelism have advantages in terms of preserving the semantics of a parallel program. On the other hand, exploiting *data parallelism* has additional complexity due to the need for morphing the graph to create multiple copies of an operator and to re-establish the order between tuples. Pipeline and task parallelism do not require morphing the graph and preserve the order without additional effort. These two forms of parallelism can be exploited by inserting the right number of threads into the data flow graph at the right locations. It is desirable to perform this kind of parallelization in a transparent manner, such that the applications are developed without explicit knowledge of the amount of parallelism available on the platform. We call this process *auto-pipelining*.

There are several challenges to performing effective and transparent auto-pipelining in the context of stream processing applications.

First, optimizing the parallelization of stream processing applications requires determining the relative costs of operators. The prevalence of user-defined operators in real-world streaming applications [69] means that cost modeling, commonly applied in database systems [66], is not applicable in this setting. On the other hand, profile-driven optimization that requires one or more profile runs based on compiler-generated instrumentation [70, 85], while effective, suffers from usability problems and lack of runtime adaptation. On the usability side, requiring profile runs and specification of additional compilation options has proven to be unpopular among users in our own experience. In terms of runtime adaptation, the profile run may not be representative of the final execution. In summary, a light-weight dynamic profiling of operators is needed in order to provide effective and transparent auto-pipelining.

Second, and more fundamentally, it is a challenge to efficiently (time-wise) find an effective (throughput-wise) configuration that best utilizes available resources and harnesses the inherent parallelism present in the streaming application. Given N operator ports and up to T threads, there are combinatorial possibilities, $\sum_{k=0}^{T} \binom{N}{k}$ to be precise. In the absence of auto-pipelining, we have observed application developers struggling to insert threads manually[1] to improve throughput. This is no surprise, as for a medium size application with 50 operators on an 8-core system, the number of possibilities reach multiple billions. Thus, a practical optimization solution needs to quickly and automatically locate an effective configuration at runtime.

Finally, deciding the right level of parallelism is a challenge. The behavior of the system is difficult to predict for various reasons. User-defined operators can contain locks that inhibit effective parallelization. The overhead imposed by adding an additional thread in the execution path is a function of the size of the tuples flowing through the port. The behavior of the operating system scheduler can not be easily modeled and predicted. The

[1] SPL language [69] provides a configuration called 'threaded port' that can be used to manually insert threads into a data flow graph.

impact of these and other system artifacts are observable only at runtime and treated as a blackbox. While the optimization step can come up with threading configuration changes that are expected to improve performance, such decisions need to be tried out and dynamically evaluated to verify their effectiveness. As such, we need a control algorithm that can backtrack from bad decisions.

In this chapter we describe an auto-pipelining solution that addresses all of these challenges. It consists of:

- A light-weight run-time profiling scheme that uses a novel metric called *per-port utilization* to determine the amount of time each thread spends downstream of a given operator input port.
- A greedy optimization algorithm that finds locations in the data flow graph at which inserting additional threads helps eliminate the system bottleneck and improve the throughput.
- A control algorithm that decides when to stop inserting additional threads and also backtracks from decisions that turn out to be ineffective.
- Runtime mechanics to insert/remove threads while maintaining lock correctness and continuous operation.

We implemented our auto-pipelining solution on IBM's System S [82] — an industrial strength stream processing middleware. We evaluate its effectiveness using micro-benchmarks, synthetic workloads, and real-world applications. Our results show that auto-pipelining provides better throughput compared to hand-optimized applications at no cost to application developers.

4.2 Background

We provide a brief overview of the basic concepts associated with stream processing applications, using SPL [69] as the language of illustration. We also describe the fundamentals

of runtime execution in System S.

4.2.1 Basic concepts

Listing 4.1 gives the source code for a very simple stream processing application in SPL, with its visual representation depicted in Figure 18 below.

Listing 4.1: `SensorQuery`: A simple application in SPL.

```
composite SensorQuery {
 type
  Location = tuple<float32 x, float32 y>;
  Sensor = tuple<uint64 sid, float64 value, Location sloc>;
  Query = tuple<uint64 qid, Location qloc, float64 radius>;
  Result = Sensor, Query, tuple<float32 distance>;
 graph
  stream<Sensor> Sensors = SensorSource() {}
  stream<Query> Queries = QuerySource() {}
  stream<Result> Results = Join(Sensors as S; Queries as Q) {
   window Sensors: sliding, time(10.0);
          Queries: sliding, count(0);
   param match: distance(S.sloc, Q.qloc) <= Q.radius;
   output Results: distance = distance(S.sloc, Q.qloc);
  }
  () as Sink = TCPSink(Results) {
   param
     role : client;
     address : "192.168.0.10";
     port : 40000;
  }
}
```

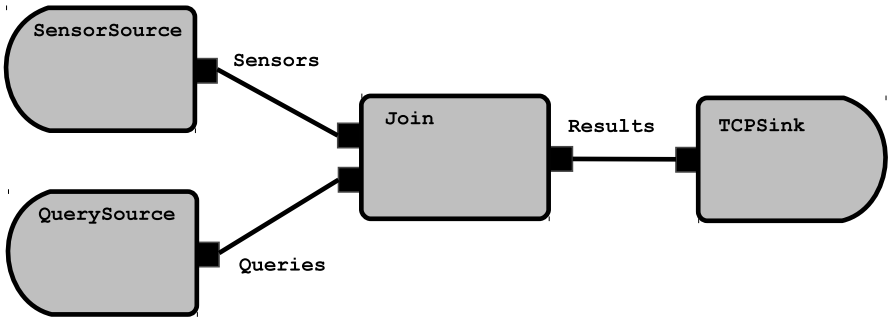


Figure 18: Data flow graph for the `SensorQuery` app.

The application is composed of *operator instances* connected to each other via *stream connections*. An operator instance is a vertex in the application graph. An operator instance is always associated with an *operator*. For instance, the operator instance shown in the middle of the graph in Figure 18 is an instance of a `Join` operator. In general, operators can have many different instantiations, each using different stream types, parameters, or other configurations such as windows. Operator instances can have zero or more input

and output *ports*. Each output port generates a uniquely named *stream*, which is a sequence of tuples. Connecting an output port to the input of an operator establishes a *stream connection*. Operators are often implemented in general purpose languages, using an event driven interface, by reacting to tuples arriving on operator input ports. Tuple processing generally involves updating some operator-local state and producing result tuples that are sent out on the output ports.

There are two important aspects of real-world applications that are highly relevant for our work:

- Real-world applications are usually much larger in terms of the number of operators they contain, reaching hundreds or even thousands.
- Real-world applications contain many user-defined re-usable operators to implement cross-domain or domain-specific manipulations.

The former point motivates the need for automatic parallelization, whereas the latter motivates the need for dynamic profiling.

4.2.2 Execution model

A distributed stream processing middleware, such as System S, executes data flow graphs by partitioning them into basic units called *processing elements*. Each processing element contains a sub-graph and can run on a different host. For small and medium-scale applications, the entire graph can map to a single processing element. Without loss of generality, in this chapter we focus on a single multi-core host executing the entire graph. Our auto-pipelining technique can be applied independently on each host when the whole application consists of multiple, distributed processing elements.

There are two main sources of threading in our streaming runtime system, which contribute to the execution of the data flow graphs. The first one is *operator threads*. Source operators, which do not have any input ports, are driven by a separate thread. When a source

operator makes a submit call to send a tuple to its output port, this same thread executes the rest of the downstream operators in the data flow graph. As a result, the same thread can traverse a number of operators, before eventually coming back to the source operator to execute the next iteration of its event loop. This behavior is because the stream connections in a processing element are implemented via function calls. Using function calls yields fast execution, avoiding scheduler context switches and explicit buffers between operators. We refer to this optimization as *operator fusion* [85, 70]. Non-source operators can also create operator threads, but this is rare. In general, the number and location of operator threads are not flexible because they are dictated by the application and the operator implementations.

The second source of threading is *threaded ports*. Threaded ports can be inserted at any operator input port. When a tuple reaches a threaded port, the currently executing thread will insert the tuple into the threaded port buffer, and go back to executing upstream logic. A separate thread, dedicated to the threaded port, will pick up the queued tuples and execute the downstream operators. Threaded port buffers are implemented as cache-optimized concurrent lock-free queues [74].

The goal of our auto-pipelining solution is to automatically place threaded ports at operator input ports during run-time, so as to maximize throughput.

4.3 System Overview

In this section we give an overview of our auto-pipelining solution. Figure 19 depicts the functional components and the overall control flow of the solution. It consists of five main stages that run in a continuous loop until a termination condition is reached.

The first stage is the *profiling stage*. In this stage a light-weight profiler determines how much time each of the existing threads spend on executing the operators in the graph. This profiling information, termed *per-port utilization*, is used as input to the *optimization* stage. An optimization algorithm that uses a greedy heuristic determines what the next action should be. The next action could either be to halt, as it could find nothing but an empty

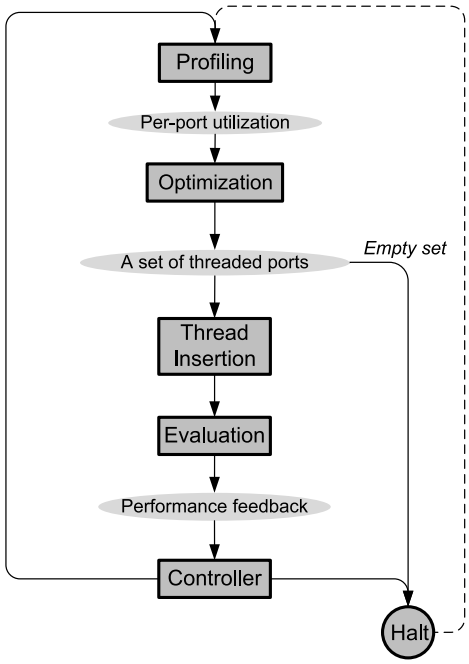


Figure 19: Overview of the auto-pipelining system.

set of threaded ports at this time, or it could be to add additional threads at specific input ports. If the optimizer decides to add new threads, then the *thread insertion* component applies this decision. This is followed by the *evaluation* component, which evaluates the performance of the system after the thread insertions. The performance results from the evaluation are put into the *controller* component as a feedback, which takes one of two possible actions. It could vet all the thread insertions and go to the next iteration of the process. Alternatively, it could remove some or all of the inserted threads, reverting back the decisions taken by the optimizer. This could be followed by moving to the next iteration of the process or halting the process. In the former case, it applies a *blacklisting* algorithm to avoid coming up with the same ineffective configuration in the next iteration.

The system can be taken out of the halting state in case a shift in the workload conditions is detected. However, the focus of this work is on finding an effective operating point right after the application launch.

4.3.1 An Example Scenario

Throughout the chapter we use an example application to illustrate various components of our solution. The compile-time and run-time data flow graphs for this application are given in Figures 20 and 21, respectively. For simplicity of exposition, we assume that all operators have a single input port and a single output port. However, our solution trivially extends to the general case and has been implemented and evaluated for the multi-port scenario (see Section 4.8).

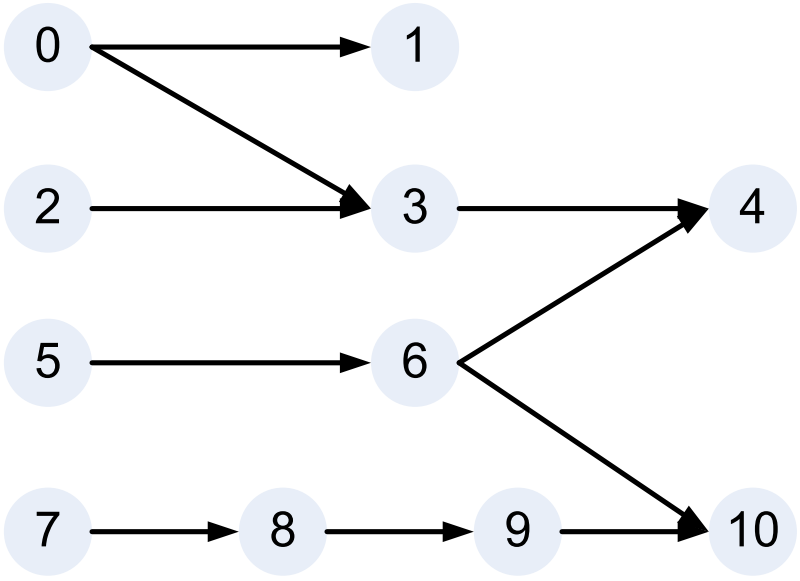


Figure 20: Operator graph

The sample application consists of an 11-operator graph as shown in Figure 20. There are four source operators (namely, o_0, o_2, o_5 and o_7) which generate tuples. At runtime, there are four threads initially, $t_0, \ldots, t_3$, that execute the program, assuming no threaded ports have been inserted. Figure 21 shows the execution path of different threads in different colors and shapes. Note that some operators are present in the execution path of multiple threads. For instance, threads t_0 and t_1 share operator o_3 in their execution paths.

The runtime graph cannot be derived solely from the compile-time graph. The paths threads take can depend on tuple runtime values, as well as operator runtime behavior, such as selectivity or tuple submission decisions. Hence, the compile-time graph restricts each thread in terms of what operators it can traverse, but does not exactly define its path. We

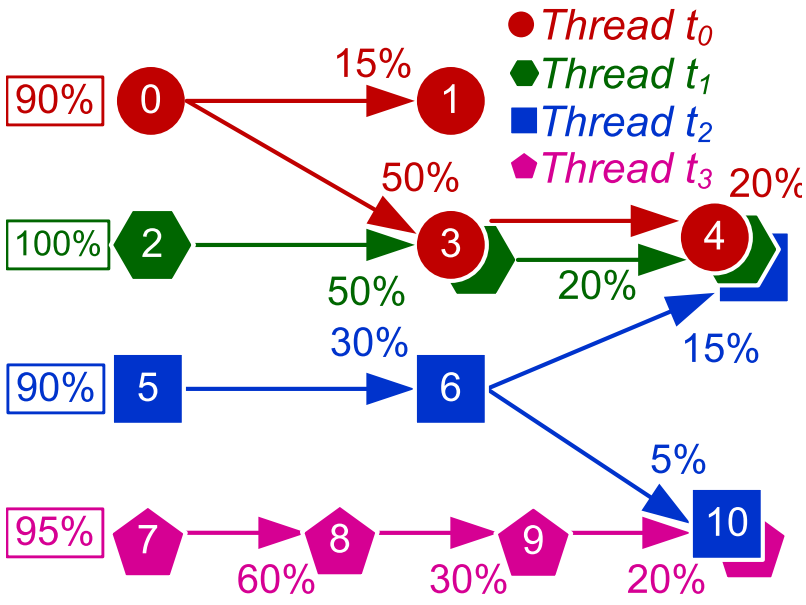


Figure 21: Runtime operator graph

derive the runtime graph based on runtime profiling (see Section 4.6).

We now look at the metrics that will help us formulate the auto-pipelining problem.

4.3.1.1 Profiling Metrics

The main profiling metric collected by our auto-pipelining solution is called the *per-port utilization*, which we denote with $\mu(o, t)$. The variable o represents any arbitrary operator, and t represents any thread that can execute that operator. We define the utilization, $\mu(o, t)$, to be the amount of CPU utilized by thread t when executing *all* downstream operators starting from the input port of operator o. During program execution, the profiler maintains $\mu(o, t)$ for every operator/thread pair for which the thread t executes the operator o. In Figure 21, for example, the input port of operator o_6 is associated with utilization 30%, meaning that thread t_2 spends 30% of the CPU time on executing o_6 and its downstream operators, which are operators o_4 and o_{10}. Thus $\mu(o_6, t_2) = 0.3$.

For each thread, we also define *per-thread utilization*, denoted as $\mu(t)$, which is the overall CPU utilization of thread t. For example, in Figure 21, thread t_2 has a utilization of 90%, thus $\mu(t_2) = 0.9$.

The reason we pick per-port CPU utilization, $\mu(o, t)$, as our main profiling metric is that

it simplifies predicting the relative work distribution between threads after inserting a new thread on an input port. For instance, if a threaded port is being added in front of operator o_6 in Figure 21, the newly created thread will take 30% CPU utilization from the existing thread t_2.

Predicting the relative work distribution for a potential thread insertion is performed in the following way. Assume that $T(o) = \{t \,|\, \mu(o,t) > 0\}$ denotes the list of threads that contain a given operator o in their execution path. Adding a threaded port at operator o will have two consequences. First, all of the threads in $T(o)$ will execute only up to the input port of operator o. Second, a new thread, t', will execute the rest of the executions paths for all threads in $T(o)$. The prediction of the work distribution for the newly created thread t' is $\mu'(t') = \sum_{t \in T(o)} \mu(o,t)$. For an existing thread $t \in T(o)$, the prediction is $\mu'(t) = \mu(t) - \mu(o,t)$. For instance, in Figure 21, when a threaded port is added to operator o_3, we predict $\mu'(t_0) = 0.4$, $\mu'(t_1) = 0.5$, and $\mu'(t') = 1$.

It is important to note that μ' is a relative metric of how the work is partitioned between the existing threads and the newly created thread. It is *not* an accurate prediction of what the CPU utilizations will be after the thread insertion. The expectation is that, given enough processing resources and enough work present in the application, the actual utilizations (μ) will be higher than the relative predictions (μ'). For instance, consider a simple chain of operators executed by a single thread that has $\mu(t_0) = 1$. Adding a threaded port in the middle of this chain will result in $\mu'(t_0) = 0.5$ and $\mu'(t_1) = 0.5$. We use these relative utilization values to assess whether or not inserting a new thread in this location will improve performance. After the insertion, the optimistic expectation is that $\mu(t_0) = \mu(t_1) > 0.5$, because $u'(t_0) < 1$ and $u'(t_1) < 1$, which leaves room for improvement in throughput. The evaluation and control stages of our solution deal with cases where this expectation does not hold.

4.3.1.2 *Utility Function*

The predicted relative utilizations are used to define a utility function that measures a threaded port insertion's goodness. Given an insertion at operator o, causing the creation of thread t', we define its *utility* as

$$U(o, t') = max(\mu'(t) \mid t \in T(o) \cup \{t'\}).$$

The utility function for a given operator and its new thread is the *largest* predicted relative work distribution across all of the threads with that operator in its path. Our goal is to *minimize* this utility function. The intuition behind the utility function is simple: the thread that has the highest predicted work (μ') will become the bottleneck of the system.

Suppose $T(o) = \{t_0\}$ and our predictions after insertion of a new thread t' at operator o are $\mu'(t_0) = 0.3$ and $\mu'(t') = 0.6$. The utility of this insertion is $U(o, t') = max(0.6, 0.3) = 0.6$. A better insertion at a different operator o', where $T(o') = \{t_0\}$, that would give a lower utility value is: $\mu'(t_0) = 0.5$ and $\mu'(t') = 0.5$, leading to $U(o', t') = 0.5$. However, it may not always be possible to find such an insertion based on the per-port utilizations of the operators reported by profiling.

For a set of thread insertions, say $C = \{\langle o, t' \rangle\}$, we define an *aggregate utility function* $U(C)$ as:

$$U(C) = max(U(o, t') \mid \langle o, t' \rangle \in C).$$

Here, we pick the maximum of the individual utilities. We will further discuss and illustrate the aggregate utility function shortly.

4.3.2 The Optimization Problem

Recall that the goal of the optimization stage is to find one or more threaded ports that will improve the throughput of the system. We propose the following heuristic for the optimization stage:

Minimize the aggregate utility function while making sure that one and only one threaded port is inserted in the execution path of each heavily utilized thread.

This formulation is based on three core principles:

1) Help the needy: At each step, we only insert threads in the execution path of heavily utilized threads. A heavily utilized thread is a bottleneck, which implies that if it has more resources, overall throughput will improve.

2) Be greedy but generous: By definition our solution is greedy, as at each step it comes up with incremental insertions that will improve performance. However, inserting a single thread at a time does not work, which is why we make sure that a thread is inserted in the execution path of each heavily utilized thread. To see this point, consider the scenario where two threads execute a simple chain of four equal sized operators. The first thread executes the first two operators, and the second thread executes the remaining two. As an incremental step, if we only help the first thread, we will end up having three threads, where the last thread still executes two operators. This imbalance will become the bottleneck and thus the throughput will not increase. But, if we help both of the two original threads, we expect the throughput to increase.

A more subtle, but critical, point is the requirement that one and only one thread is added to the execution path of each heavily utilized thread. This is strongly related to the greedy nature of the algorithm. If we are to insert more than one thread in the execution path of a given thread, then the prediction of a thread's μ' requires significantly more profiling information (such as the amount of CPU time a thread spends downstream of a port when it reaches that port by passing through a given set of upstream input ports). We want to maintain a light-weight profiling stage that will not disturb application performance during profiling. Hence, we make our algorithm *greedy* by inserting at most one thread in the execution path of an existing thread, but for each one of the heavily utilized threads (thus *generous*).

3) Be fair: We minimize the utility function U, which means that new threads are inserted such that the newly created and the existing threads have balanced load.

4.4 Optimization Algorithm

We now describe a base optimization algorithm and a set of enhancements that improve its running time.

4.4.1 The Algorithm

For a simple chain of operators, designing an algorithm that meets the criteria given in Section 4.3.2 is straightforward. However, operators that are shared across threads complicate the design in the general case. We need to make sure that one and only one thread is inserted in the execution path of each existing thread, even though the same thread can be inserted in the execution path of multiple existing threads. The main idea behind the algorithm is to reduce the search space via selection and removal of shared operators from the set of possible solutions, and then explore each sub-space separately.

Before describing the algorithm in detail, we first introduce a simple matrix form that represents a sub-space of possible solutions.

Matrix representation For each thread, we initially have all the operators in the execution path of it as a possible choice for inserting a threaded port. As the algorithm progresses, we gradually remove some of the operators from the list to reduce the search space. For instance, the runtime operator graph from Figure 21 can be converted into the following matrix representation:

$$
\begin{array}{c}
\\
t_0 \\
t_1 \\
t_2 \\
t_3
\end{array}
\left(\begin{array}{c|ccc}
o_0, 90\% & o_1, 15\% & o_3, 50\% & o_4, 20\% \\
o_2, 100\% & o_3, 50\% & o_4, 20\% & \\
o_5, 90\% & o_6, 30\% & o_4, 15\% & o_{10}, 5\% \\
o_7, 95\% & o_8, 60\% & o_9, 30\% & o_{10}, 20\%
\end{array}\right)
$$

The matrix contains one row for each thread in the unmodified application. For each row, it lists the set of operators that are in the execution path of the thread with their associated CPU utilization metrics, which is $\mu(o, t)$. Note that the source operators are placed on the first column and are separated from the rest. They are not considered as potential places to add threaded ports as they have no input ports. We exclude them from the matrix representation for the rest of the chapter. The remaining operators are in no particular order, but we sort them by their index for ease of exposition.

The algorithm is composed of four major phases, namely *bottleneck selection*, *solution reduction*, *candidate formation*, and *solution selection*.

Bottleneck Selection The first phase is the bottleneck selection, which identifies highly utilized threads. A threshold $\beta \in [0, 1]$ is used to eliminate threads whose CPU utilizations are below it. For instance, if $\beta = 0.92$, threads t_0 and t_2 are eliminated since their utilizations are smaller than the threshold and thus are not deemed bottlenecks. For the rest of this section, we assume $\beta = 0.8$ for the running example, which means all of the four threads are considered as bottlenecks.

Solution Reduction The second phase is the solution reduction, which performs a tree search to reduce the solution space. At the root of the tree is the initial matrix. At each step, we choose one of the leaf matrices that still contains shared operators based on the runtime data flow graph. We pick one of these shared operators for that leaf matrix and perform *selection* and *removal* to yield two sub-matrices in the tree.

Selection means that we select the shared operator as part of the solution, and thus remove all other operators from the rows that contain the shared operator. Furthermore, we remove all operators that originally appeared together with the shared operator in the same row, from other rows, since they cannot be selected in a valid solution. Figure 22 shows an example. Consider the edge labeled $S3$, which represents the case of selecting shared operator 3. After the selection, the first two rows now have operator 3 as the only choice.

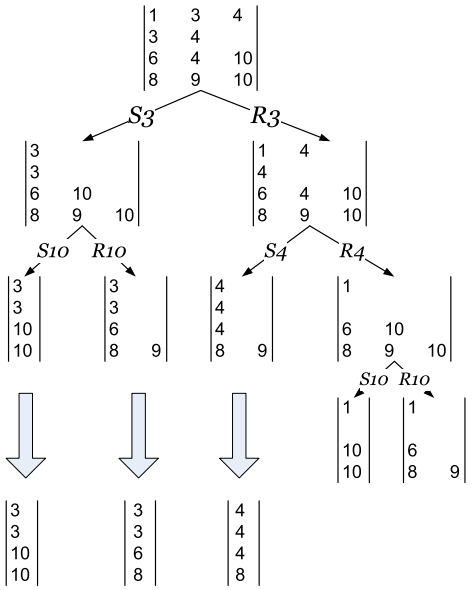


Figure 22: Solution reduction and candidate formation.

Furthermore, operators 1 and 4—which previously appeared in the same row as 3—are removed from all rows, as picking them would result in inserting more than one thread on the execution paths of the first two threads.

Removal means we exclude the shared operator from the solution, and thus we remove it from all rows where it appears. Figure 22 shows an example. Consider the edge labeled $R3$, which represents the case of removing the shared operator 3.

The solution reduction phase continues until the leaf matrices have all of their shared operators removed.

Candidate Formation After the solution reduction phase, all leaves of the tree contain *pre-candidate* solutions. The goal of the candidate formation phase is to create candidate solutions out of the pre-candidate ones. As part of candidate formation, first we apply a filtering step. If we encounter a leaf matrix where a thread is left without an operator in its row, yet there is another dependent thread that has a non-empty row, then we eliminate this leaf matrix. We consider threads that share operators in their execution paths as dependent.

As an example, the rightmost two leaves in Figure 22 are removed in the filtering step. After the filtering step, we convert each remaining pre-candidate solution into a candidate

solution by making sure that each non-empty row contains a single operator, i.e., we convert each matrix into a column vector. When there are multiple operators in a row, we compute the utility function $U(o,t)$ for each, and pick the one that gives the lowest value. As an example, the pre-candidate solution pointed at by arrow $S4$ in Figure 22 is converted into a candidate solution by picking operator 8 as opposed to operator 9. Operator 8 has a lower utility value, $U(o_8,t_3) = 0.6$, compared to operator 9's utility, $U(o_9,t_3) = 0.65$.

Solution Selection In the solution selection phase, we pick the best candidate among the ones produced by the candidate formation phase. Recall that our utility function $U(o,t)$ was defined on a per-thread basis. To pick the best candidate, we use the aggregate utility function $U(C)$, where $C = \{\langle o,t\rangle\}$ represents a candidate solution. Recall that we pick the maximum of the individual utilities[2], thus $U(C) = max(U(o,t) \,|\, \langle o,t\rangle \in C)$. We pick the candidate solution with the minimum aggregate utility as the final solution. In the running example, this corresponds to picking $C = \{\langle o_4,t_0\rangle, \langle o_4,t_1\rangle, \langle o_4,t_2\rangle, \langle o_8,t_3\rangle\}$ with aggregate utility of 0.8.

4.4.2 Complexity Analysis

Here, we provide a cost analysis of the base algorithm.

Suppose there are initially r threads, which is equal to the number of rows in the initial solution matrix. Further assume that there are s shared operators in the runtime graph. To analyze the complexity of our algorithm, we start by calculating the number of leaf candidates, denoted as $cn(s,r)$, in the solution reduction phase. $cn(s,r) = \sum_x cn(s,r,x)$, where $cn(s,r,x)$ denotes the number of candidates with x shared operators in them. $cn(s,r,x)$ must be smaller than the cardinality of x choose s, that is $cn(s,r,x) \leq \binom{s}{x}$. Note that a single candidate can have up to $\frac{r}{2}$ shared operators in it as each shared operator fixes the assignments for at least two rows of the solution matrix. Therefore,

[2]When there are more than one dependent thread groups, utility U is computed independently for each group and the maximum is taken as the final aggregate utility.

$$
\begin{aligned}
cn(s,r) &= \sum_{x\in[0,\min(s,\frac{r}{2})]} cn(s,r,x) \\
&\leq \sum_{x\in[0,\min(s,\frac{r}{2})]} \binom{s}{x} \\
&= \begin{cases} \Theta(2^s) & \text{if } \frac{r}{2} \geq \frac{s}{2} - \sqrt{s}, \\ \Theta\Big[\frac{\binom{s}{r/2}}{\left(1-r/s\right)}\Big] & \text{otherwise.} \end{cases}
\end{aligned} \tag{13}
$$

The last step is due to a closed form for partial sum of binomial coefficients [83]. $cn(s,r)$ can be used as a rough bound for algorithm complexity, assuming each candidate costs $O(1)$ computation units. The complexity of the algorithm can be further bounded by considering that the solution reduction operates independently for sets of initial threads that are disjoint in terms of their operators. Thus we can represent the number of candidates as $\sum_{\langle s_i,r_i\rangle} cn(s_i,r_i)$ where $\sum_i r_i = r$ and s_i represents the number of shared operators in the ith thread group. For large graphs, $max(\{r_i\})$ is often smaller that r. However, this does not change the worst case complexity.

We can provide a finer-grained complexity analysis by breaking down the per-candidate cost. First, we use m to denote the number of operators that appear in the initial solution matrix. The algorithm finds the set of shared operators in a two-level nested loop, in which case each loop is a scan of the whole matrix, costing $O(m^2)$ units. During the solution reduction phase, $cn(s,r)$ leaf candidates in the search tree implies $cn(s,r)-1 = O(cn(s,r))$ internal nodes. The split operation at each internal node needs a full scan of the matrix, resulting in a cost of at most m. The candidate formation phase needs to scan the matrix of each candidate, thus leading to per-candidate cost of at most m. The last phase, solution selection, costs r units per candidate since each candidate is a column vector. In total, the complexity is bounded by $O(m^2 + cn(s,r)\cdot(2m+r))$, which further simplifies to $O(m^2 + cn(s,r)\cdot m)$.

As we will see later in Section 4.8, the algorithm runs quite fast in practice for large

graphs, especially when the the pruning optimization is applied.

4.4.3 Algorithm Enhancements

We further propose and employ two enhancements to our basic algorithm.

Pruning: Our enhanced algorithm stops branching when it finds that the utility function value for some of the rows in the current matrix is already equal to or larger than 100%. For example, in Fig. 22, there is no point to continue branching after $R4$, since thread t_1 has no potential threaded port to add and will remain bottlenecked after inserting other threads.

Sorting: In the solution reduction phase, we use the degree of operator sharing as our guideline for picking the next solution to further reduce. We sort the shared operators based on the number of rows they appear in. This way, if a shared operator shows up in the execution path of many threads, it is considered earlier in the exploration as it will result in more effective reduction in the search space, especially when used with pruning. When selected, shared operators have a higher chance of causing the utility function value to go over 100% due to contribution from multiple threads.

4.5 Evaluation and Control

The thread insertions proposed by the optimization stage are put into effect by the runtime. After inserting the new threads, the evaluation stage measures the throughput on input ports which received a threaded port. The throughput is defined as the number of tuples processed per second. If the throughput increased for all of the input ports that has received a threaded port, then the controller stage moves on to the next iteration.

If the throughput has not increased for some of the input ports, then the control stage performs *blacklisting*. The ports for which the throughput has not improved are blacklisted. Furthermore, the thread insertions are reverted by removing these threaded ports from the flow graph. Blacklisted input ports are excluded from consideration in future optimization stages. If the percentage of blacklisted input ports exceeds a pre-defined threshold $\alpha \in [0, 1]$, then the process halts. Otherwise, we move on to the next iteration. It is possible that

the process halts even before the threshold α is reached, as a feasable solution may not be found during the optmization stage.

Alternative blacklisting policies can be applied to reduce the change of getting stuck at a local minima. For instance, the blacklisted ports can be maintained on a per-pipelining configuration basis rather than globally, at the cost of keeping more state around.

4.6 Profiler

We describe the basic design of the profiler component.

Our profiler follows the design principle of *gprof* [76], that is, to use both instrumentation and periodic sampling for profiling. However, the instrumentation is not part of the generated code. Instead, the SPL runtime has lightweight instrumentation which records thread activity with respect to operator execution. More specifically, the instrumented SPL runtime monitors the point at which a thread enters or exits an input port, so that it can track which ports are currently active. It uses a special per-thread stack, called the E-stack, for this purpose.

In order to collect the amount of CPU time a thread spends downstream of an input port, our system periodically samples the thread status and traverses the E-stacks. We call the period between two consecutive samplings the *sampling period*, denoted by p_s. If there are N occurrences during the last p_o seconds where thread t was found to be active doing work downstream of operator o's input port, then the per-port thread utilization $\mu(o, t)$ is given by $\frac{N}{p_o/p_s}$. The intuition for this calculation is that it is the number of observations (N) divided by how many times we sample during a given time period (p_o/p_s).

Periodic sampling is inherently subject to statistical inaccuracy, thus enough samples should be collected for accurate results. This could be achieved by either increasing the duration of profiling (p_o) or decreasing the sampling period (p_s). Given the long running nature of streaming applications, we favor the former approach.

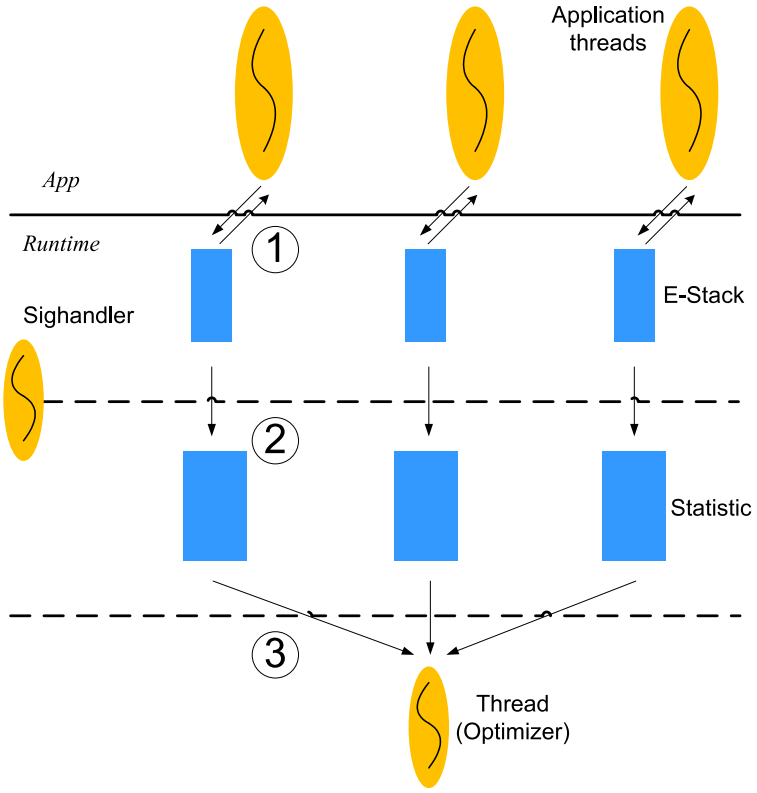


Figure 23: Profiler implementation

4.6.1 Implementation notes

Our runtime profiler implementation consists of three components. Figure 23 illustrates each component, which we also describe below.

E-stack For each application thread, the profiling system maintains a simple execution stack, called an *E-stack*. The system pushes/pops entries into/from the E-stack every time the thread associated with the stack enters/exits an operator. Figure 24 shows a snapshot of the E-stack of a thread executing operator o_{10} after going through operators o_5 and o_6.

Signal handler For each sampling period p_s, the profiling system checks the execution stacks of all actively running threads. This is achieved by registering a timer for the signal `SIGPROF` with the timer interval set to p_s. The operating system then sends signal `SIGPROF` every p_s seconds, which one of the application threads receives inside of a signal handler. Upon each receipt of the signal, the signal handler takes snapshots of the

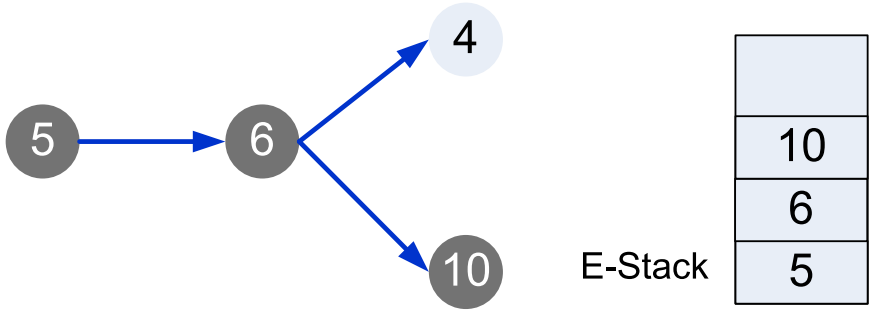


Figure 24: Example of an E-stack

E-stack for all currently active[3] threads. For each stack entry in an E-stack snapshot, it increments a counter for the thread and operator pair associated with the entry. Note that it scans the entire stack because our goal is to compute the amount of CPU time a thread spends *downstream* of a given operator input port. For instance, in Figure 24, it increments the counters for operators o_5, o_6, and o_{10}, as the active thread is doing work downstream of all of these three operators at this time.

Summarization Every optimization period p_o, the counters maintained for each thread and operator pair are summarized into the per-port thread utilization numbers. These are the final set of statistics that will be used by the optimization stage.

A problem associated with the simple counting scheme used for profiling is that, in a multi-threaded environment, the more active threads there are, the more frequently (in wall clock time) signal `SIGPROF` is delivered. This skews the statistics. We use a mechanism called *frequency autoscaling* to correct this skew. Rather than incrementing each counter by 1 at each sampling step, we increment it by $1/A$ where A is the number of currently active threads.

To handle profiling in a multi-threading environment, the consistency problem arises when E-stacks are written by SPL runtime while being read by the signal handler. HProf [93] tackles this problem by suspending threads at every sampling, which however interrupts program execution and increases overhead. We avoid thread suspension or any

[3]The `proc` system in Linux is used for this purpose in the current prototype.

locking mechanism by writing E-stack entries in an atomic yet efficient way.

Each stack entry is stored as a 64-bit volatile value in a 64-bit system — a 32-bit operator identifier and a 32-bit port index. We rely on the details of the Intel architecture to ensure that reading the stack entries is atomic. The current size of the stack is also kept as a volatile counter. Since the updating of the stack entries and the stack size are not transactional, sometimes the profiler can scan an entry that is not active. However, since this happens with very low frequency it barely impacts the aggregate values computed by the profiler.

4.7 Dynamic Thread Insertion/Removal

Thread insertion and removal is implemented by dynamically adding and removing threaded ports. Both activities require suspending the current flow of data for a very brief amount of time, during which the circular buffer associated with the threaded port is added/removed to/from the data flow graph. Finally, the suspended flow is resumed. Suspending the flow, however, is not the only step necessary to preserve safety. In the presence of *stateful* operators, dynamic lock insertion and removal is required to ensure mutually exclusive access to shared state, as elaborated below. Our implementation does make use of thread pools, since the additional work that is performed during thread injection and removal dominates the overall cost.

4.7.1 Locking & Thread Insertion/Removal

There is a subtle concern regarding thread safety in the presence of thread insertion/removal and multi-threaded execution of *stateful* operators. Before we describe the potential problems that may result from thread insertions, we first give a brief overview of the relevant aspects of the programming model used to develop operators in SPL.

Operators are implemented through an event-driven interface as described in Section 4.2.1. An operator that contains state which is modified as a result of processing tuples delivered to one of its input ports is said to be stateful. Such operators need to ensure that

the state is protected against concurrent modification. Recall that operators can be executed concurrently by multiple threads. In SPL, stateful operators use an *auto port mutex* object to protect their state from concurrent modification. An auto port mutex is a scoped mutex that either creates a critical section around a block of code, or simply reduces to a no-op at the cost of an untaken branch.

The SPL runtime decides which one of these behaviors is to be exhibited depending on safety analysis. Operator developers always protect their state from concurrent access using auto port mutexes, yet the runtime can decide to effectively remove these mutexes when it is safe to do so. The safety analysis is performed by a simple process of *thread propagation* to decide if an operator can potentially be called by multiple threads.

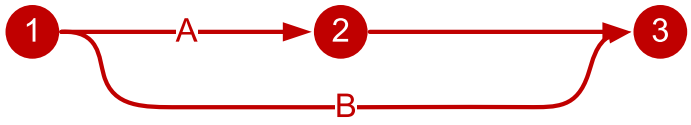


Figure 25: Thread propagation: Initial state

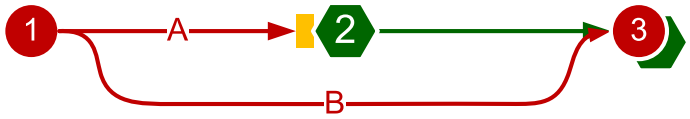


Figure 26: Thread propagation: After adding port 2

The thread propagation analysis is performed every time a thread is inserted or removed at runtime. When a thread is inserted, the analysis is needed to *turn on* some of the auto port mutexes to ensure safety. For a thread removal, it needs to be performed to *turn off* some of the auto port mutexes to ensure good performance.

Figure 25 shows an example SPL application. In this example, we want to add a threaded port to operator o_2. As a result of this change, there will be two threads executing the downstream operator o_3 as shown in Figure 26. If o_3 is a stateful operator, then the auto port mutexes used by the operator are turned on by the runtime, before the flow is resumed following the insertion.

4.8 Experimental Results

We evaluate the effectiveness of our solution based on experimental results. We perform three kinds of experiments. First, we use micro-benchmarks to evaluate the components of our solution and verify the assumptions that underlie our techniques. Second, we evaluate the running time efficiency of our optimization algorithm under varying topologies and application sizes, using synthetic applications. Third, using three real-world applications, we compare the throughput our auto-pipelining scheme achieves to that of manual optimization as well as no optimization.

4.8.1 Experimental Setup

We have implemented our auto-pipelining scheme in C++, as part of the SPL runtime within System S [82].

All of our experiments were performed on a host with 2 Intel Xeon processors. Each processor has 4 cores, and each core is a 2-way SMT, exposing 16 hardware threads per node, but only 8 independent cores. When running the experiments, we turn off hyper-threading so that the number of virtual cores equals the number of physical cores (which is 8).

4.8.2 Micro-benchmarks

For the micro-benchmarks, we use a simple application topology that consists of a chain of 8 operators. All operators have the same cost and perform the same operation (a series of multiplications). The cost of an operator is configurable.

4.8.2.1 Pipelining benefit

Pipelining is beneficial under two conditions. First, enough hardware resources should exist to take advantage of an additional thread. Second, the overhead of copying a tuple to a buffer and a thread switch-over should be small enough to benefit from the additional

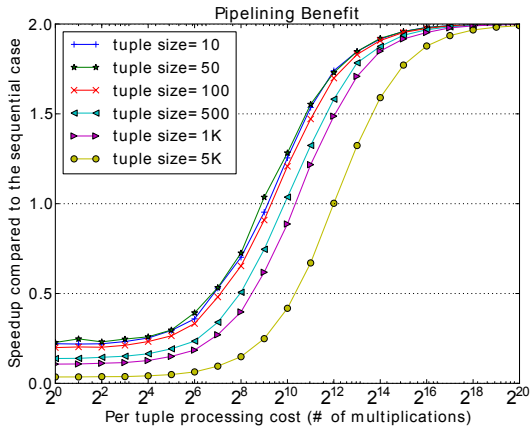


Figure 27: Speedup vs. processing cost.

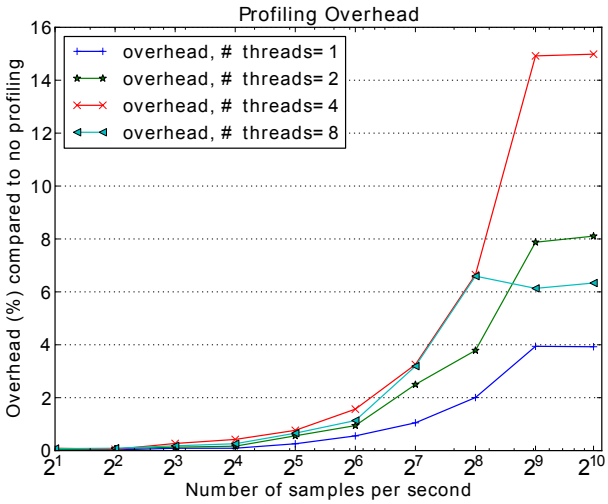


Figure 28: Profiling overhead vs. sampling rate.

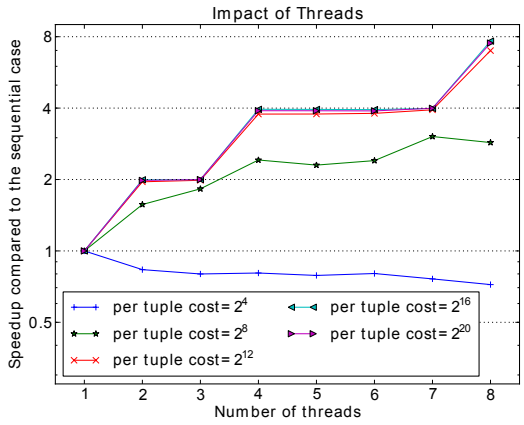


Figure 29: Speedup for different # of threads

parallelism. When these conditions do not hold, the evaluation and control stages of our auto-pipelining solution will detect this and adjust the adaptation process.

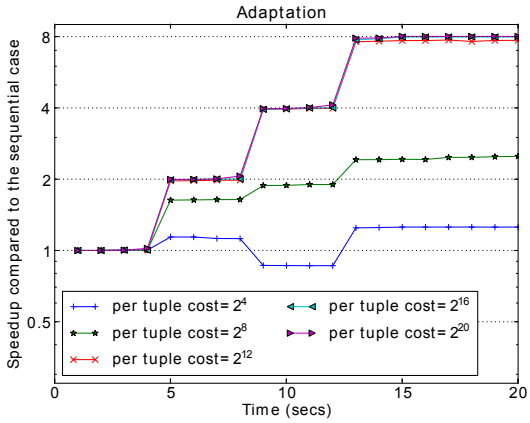


Figure 30: Adaptation with auto-pipelining

We evaluate the pipelining benefit and show how it relates to the overhead associated with threaded ports by measuring the speedup obtained when executing our application with two threads instead of one. Figure 27 plots the speedup as a function of the per-tuple processing cost, for different tuple sizes. When the per-tuple processing cost is small, it is expected that using an additional thread will introduce significant overhead. In fact, we observe that the additional thread reduces the performance (speedup less than 1). As the per-tuple processing cost gets higher, we see that perfect speedup of $2\times$ is achieved. The tuple sizes also have an impact on the benefit of pipelining. For large tuple sizes, the additional copying required to go through a buffer creates overhead. Thus, the crossover point for achieving $> 1\times$ speedup happens at a lower per-tuple cost for smaller sized tuples. For small tuples, custom allocators [123] can be used to further improve the performance. For large tuples, the copying of the data contents dominates the cost. While copy-on-write (COW) techniques can be used to avoid this cost, it is well accepted that COW optimizations are not effective in the presence of multi-threading.

4.8.2.2 Profiling overhead

Light-weight profiling that does not disturb application performance is essential for performing auto-pipelining at run-time. In Figure 28 we study the profiling overhead. The overhead is defined as the percent reduction in the throughput compared to the non-profiling

case. The figure plots the overhead as a function of the number of samples taken per second, for different number of threads. The operators are evenly distributed across threads. We observe that, as a general trend, the profiling overhead increases as the profiling rate grows. For the remainder of the experiments in this chapter, we use a profiling sampling rate of $p_s = 100$, which corresponds to a 3% reduction in performance. Note that the profiler is only run for a specific period (for p_o seconds) during one iteration of the adaptation phase. Once the adaptation is complete, no overhead is incurred due to profiling.

We further observe that increasing the profiling rate beyond a threshold does not increase the overhead anymore. This is because the system starts to skip profiling signals when the sampling period p_s is shorter than the time needed to run the logic associated with the profiling signal. Interestingly, the profiling overhead does not monotonically increase with the number of threads. At first glimpse, this may be unexpected since more threads means more execution stacks to go through during profiling. However, with more threads, each execution stack has less entries, which decreases the overhead.

For most operator graphs, it is the depth of the operator graph that impacts the worst case profiling cost, rather than the number of threads used. For instance, for a linear chain, the number of stack entries to be scanned only depends on the depth of the graph. For bushy graphs this number can also depend on the number of threads, even though it is rarely linear in the number of threads in practice (a reverse tree is the worst case).

4.8.2.3 Impact of threads

Recall that one of the principles of our optimization is to insert a threaded port in the execution path of each bottleneck thread. We do this because the speedup from adding threads one-at-a-time will result in a series of non-improvements, followed by a jump in performance when all bottleneck threads finally get help. In Figure 29, we verify this effect. The figure plots the speedup as a function of the number of threads, for different tuple costs. The threads are inserted in a balanced way, by picking the thread that executes the highest

number of operators and partitioning it into two threads.

We observe that, for sufficiently high per-tuple processing costs, the speedup is a piecewise function which jumps at certain number of threads, like 2, 4, and 8. Each such jump point corresponds to a partitioning where all threads execute the same number of operators. This result justifies our algorithm design which inserts multiple threaded ports in one round. For low per-tuple processing costs (such as 2^8) the speedup is not ideal, and for very low per-tuple processing costs (such as 2^4), the performance degrades.

4.8.2.4 Adaptation

We evaluate the adaptation capability of our solution by turning on auto-pipelining in an application whose topology is a simple chain of `Functor` operators. For this experiment, we measure the throughput of the application as a function of time. The adaptation period is set to 5 seconds. We report the throughput relative to the sequential case. Figure 30 reports these results for different per-tuple processing costs.

We observe that our algorithm intelligently achieves optimal speedup for different per-tuple costs. For instance, when the per-tuple cost is 2^4, our algorithm finds out that its second optimization decision does not improve overall throughput, and thus it rolls back to the previous state. For higher per-tuple costs, such as 2^{20}, the algorithm does not stop adding threaded ports until it reaches the unpartitionable state, that is 1 operator per thread. Comparing Figures 29 and 30, we see that auto-pipelining lands on the globally optimal configuration in terms of the throughput.

The total adaptation time of the system depends on two major components: (i) the number of steps taken, and (ii) the adaptation period. Since our algorithm helps all bottlenecked threads at each step, its behavior with respect to the number steps taken is favorable. For instance it takes $\log_2(8) = 3$ steps to reach 8 threads in Figure 30. For more dynamic scenarios, we can reduce the adaptation period to reduce the overall adaptation time. The only downside is that, reducing the adaptation period without decreasing the accuracy of the

profiling data requires increasing the profile sampling rate, which can increase the profiling cost.

4.8.2.5 *Operator cost vs. throughput*

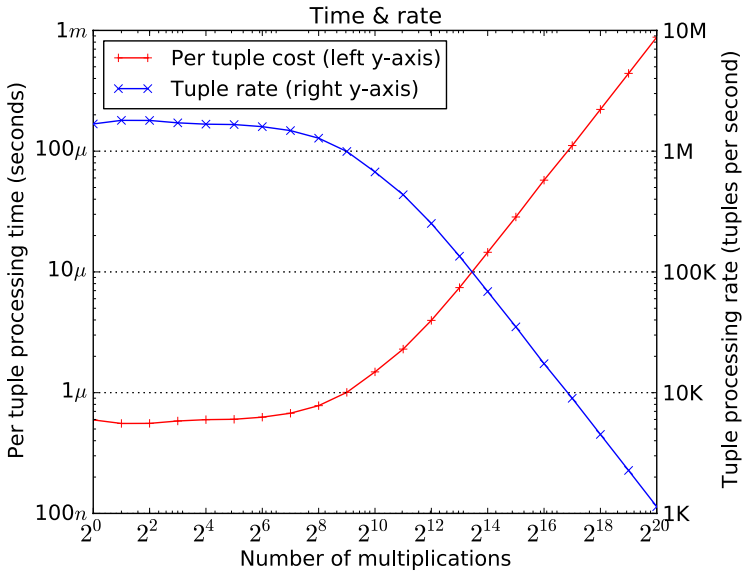


Figure 31: Per-tuple processing time and max. rate as a function of processing cost.

Figure 31 plots the time it takes to process a single tuple (left y-axis), as well as the maximum rate of processing that can be achieved with a single operator (right y-axis) as a function of the per-tuple processing cost represented as the number of multiplications.

4.8.3 Synthetic Benchmarks

This set of experiments evaluate the running-time efficiency of our optimization algorithm, as well as the effectiveness of the two enhancements, *pruning* and *sorting*.

In these experiment, we use two different kinds of synthetic topologies. These are the *reverse tree* and the *random graph* topologies. For a reverse tree topology, each leaf-level operator serves as a starting point for a different thread and each thread executes the set of operators that forms a path from the leaf-level operator to the root. We pick this topology as it represents an extreme scenario where there is massive amount of sharing across threads. In our experience, this kind of topology is not seen often in practice.

For a random topology, operator IDs are randomly picked from a finite domain of integers $[0, d)$. Here, a parameter called *sampled ratio*, denoted by sr, is used to measure the degree of overlapping between threads, that is, how many operators in one thread can be shared by another thread. Suppose there are n threads and each thread executes m operators, then $sr = \frac{n \cdot m}{d}$. Utilization values are uniformly distributed among operators for each thread.

We run each experiment 5 times and report the average performance numbers. The reported performance is the search cost, which is quantified by the number of tree nodes traversed during the execution of the optimization algorithm.

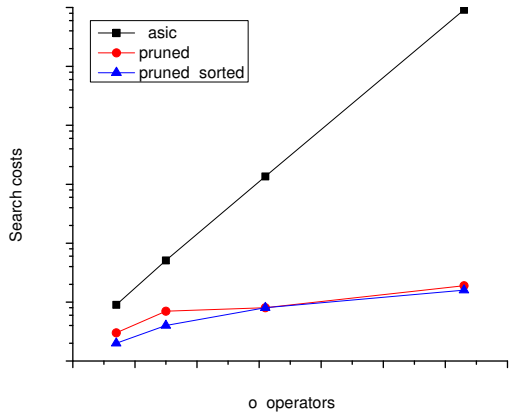
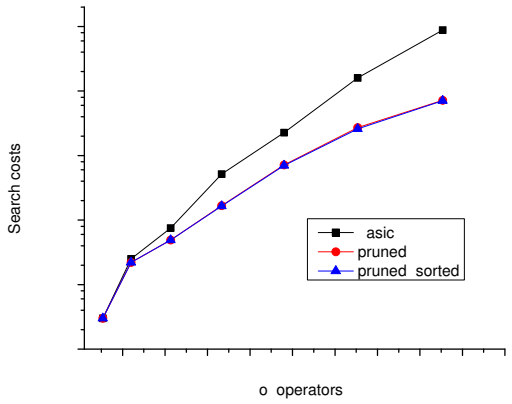


Figure 32: Performance of optimization algorithm with the reverse tree topology

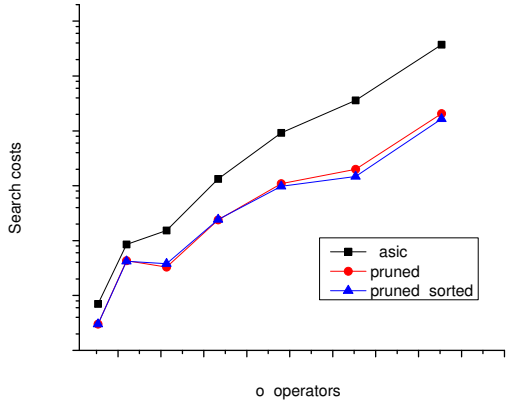
Figure 32 presents the results for the reverse tree topology, in which different tree sizes are tested from 2^3 to 2^6. As expected, search costs of our basic algorithm (i.e., without any enhancements) grow exponentially with the number of shared operators [4]. We observe that compared to the basic algorithm, pruning is very effective. It achieves an order of magnitude saving in search cost (the y-axis is in log scale). Applying sorting on top of pruning provides modest additional improvement, only for small number of operators.

Our results are further corroborated by the random graph based experiments. For these experiments, we varied two parameters: the number of operators n and the sampled ratio

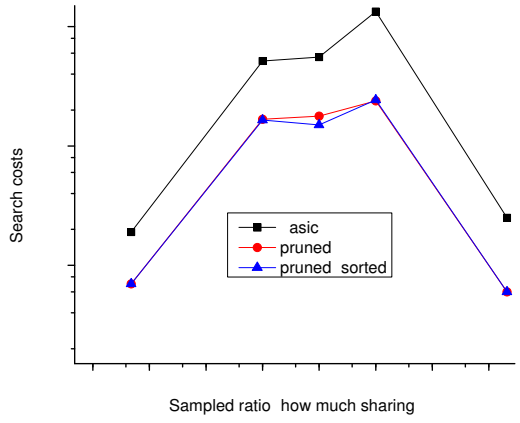
[4] In a reverse tree topology, the number of operators equals that of shared operators.



(a) Less shared threads ($sr = 0.5$)



(b) More shared threads ($sr = 2$)



(c) As a function of sampled ratio

Figure 33: Performance of optimization algorithm with random topology

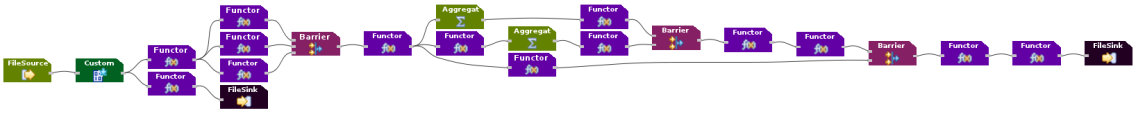


Figure 34: Lois – Cosmic ray shower detection application

sr. These results are shown in Figure 33.

Figure 33a plots the search cost as a function of number of operators with fewer sharing (the sampled ratio sr is as small as 0.5), while Figure 33b plots the search cost as a function of number of operators with more sharing (sr is set to 2). We observe that although the search costs with more sharing between threads tend to be higher, the overall trend with increasing number of operators is similar. Search costs largely grow linear with the number of operators.

Figure 33c plots the search cost as a function of the sampled ratio. We observe that the search costs first grow with increasing sampled ratios and then drop. While increased sharing raises the cost of the algorithm initially, excessive sharing results in shrinking the search space, resulting in this bi-modal behavior.

Overall, Figure 33 shows that pruning is a very effective optimization strategy for our algorithm, under various circumstances, often providing an order of magnitude improvement in running time.

4.8.4 Application Benchmarks

The application benchmarks consist of three real-world stream processing applications with their associated workloads. These applications are named Lois, Vwap, and LinearRoad. The LinearRoad application (the smallest of the three) is depicted in Figure 36, whereas other applications are depicted below.

The Lois [21] dataset is collected from a Scandinavian radio-telescope under construction in northwestern Europe. The goal of the Lois application is to detect cosmic ray showers by processing the live data received from the radio-telescope.

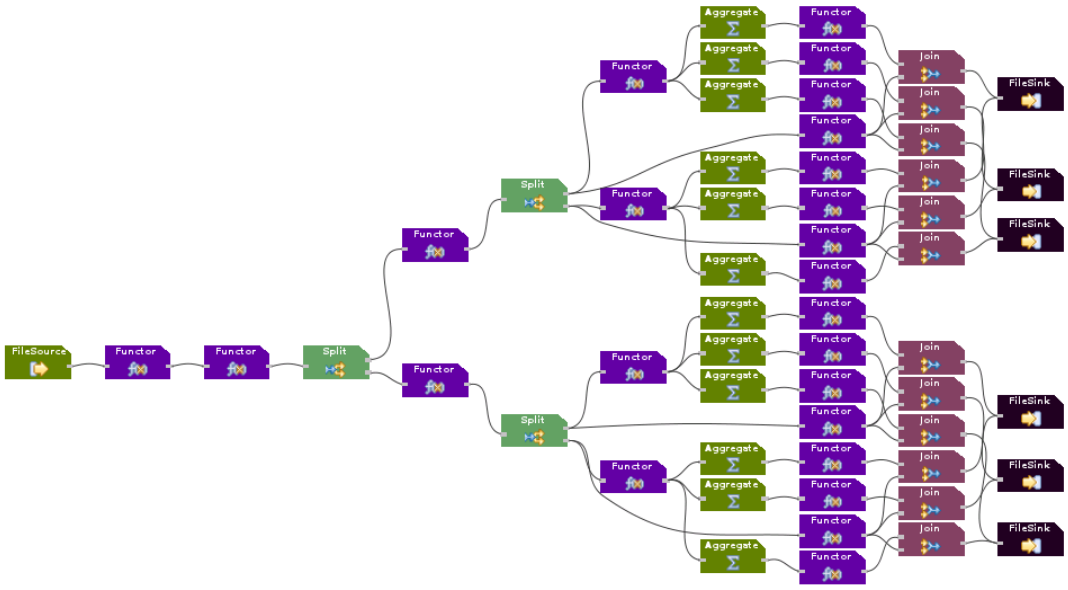


Figure 35: Vwap – Bargain detection application

The Vwap [38] dataset contains financial market data in the form of a stream of real-time bids and quotes. The goal of the Vwap application is to detect bargains and trading opportunities based on the processing of the live financial feed.

LinearRoad [39] dataset contains speed, direction, and position data for vehicles traveling on road segments. The goal of the application is to compute tolls for vehicles traveling on the hypothetical "Linear Road" highway.

4.8.4.1 Application Graphs

The Vwap and Lois applications are depicted in Figures 35 and 34, respectively. LinearRoad application is depicted in Figure 36.

Vwap is a medium-scale application (58 operators) that contains large number of Functor, Join, and Aggregate operators. The heavy joins and aggregations create opportunities for pipeline parallelism.

Lois is a small-scale application (22 operators) that contains large number of Functor operators. It contains 3 Barriers, which is indicative of task parallelism being present in the flow graph. LinearRoad is a small-scale application (15 operators) that contains large number of Custom operators. It contains 2 Unions, which is indicative of task parallelism

also being present in the flow graph.

The Lois and LinearRoad applications have few branches in their topology, whereas Vwap has many. The LinearRoad application makes heavy use of custom operators, whereas the other applications are composted of mostly built-in operators.

Table 7 gives a breakdown of the operators constituting the three applications.

Table 7: Breakdown of operators used in the Lois and Vwap applications

	Instance Count		
Operator Kind	Lois	Vwap	LinearRoad
Functor	13	24	3
Join	0	12	0
Split	0	3	2
Barrier	3	0	0
Union	0	0	2
Aggregator	2	12	0
Custom	1	0	5
FileSource	1	1	1
FileSink	2	6	1

It is important to note that the Lois and LinearRoad applications have few bush segments in their topology, whereas Vwap has many. The LinearRoad application makes heavy use of custom operators, whereas the other applications are composted of mostly built-in operators.

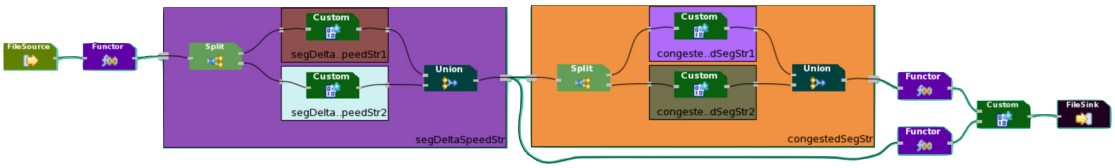


Figure 36: LinearRoad – A vehicle toll computation app.

We run three versions of these programs: unoptimized, hand-optimized, and auto-pipelined. The hand-optimized versions are created by explicitly inserting threaded ports in the SPL code of the application. This was carried out by the application developers, independent of our work. For all cases, we measure the total execution time for the entire data set. For the auto-pipelined version, the adaptation period is also included as part of the

total execution time.

Figure 37 gives the results. For the Lois application, we see around $1.5\times$ speedup compared to the unoptimized version, for Vwap we see around $3\times$ speedup, and for LinearRoad we see $2.56\times$ speedup. Note that these are real-world applications, where sequential portions and I/O bound pieces (sources and sinks) make it difficult to attain perfect speedup. It is impressive that our auto-pipelining solution matches the hand-optimized performance in the case of Lois, and improves upon it by around $2\times$ for both Vwap and LinearRoad. It is also worth noting that in the case of Lois, the programmer has statically added threaded ports based on her experience and the suggestion from a fusion optimization tool called COLA [85]. Considering that the auto-pipeliner takes around 20 seconds to adapt in this particular case, the throughout attained for the auto-pipelining solution is in fact higher than the hand-optimized case.

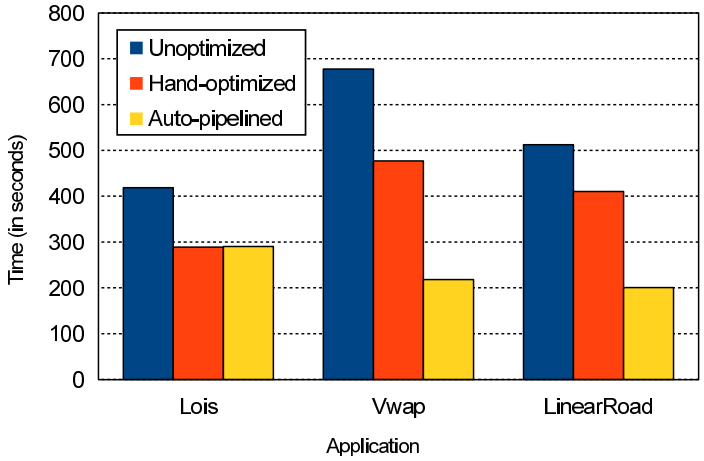


Figure 37: Running time for Lois and Vwap

Overall, auto-pipelining provides equal or significantly better performance compared to hand optimization, at no additional cost to the application developers.

4.9 Related Work

Our work belongs to the area of auto-parallelization and we survey the related topics accordingly.

Dynamic multi-threaded concurrency platforms, such as Cilk++ [2], OpenMP [26], and

x10 [54], decouple expressing a program's innate parallelism from its execution configuration. OpenMP and Cilk++ are widely used language extensions for shared memory programs, which help express parallel execution in a program at development-time and take advantage of it at run-time.

Kremlin [68] is an auto-parallelization framework that complements OpenMP [26]. Kremlin recommends to programmers a list of regions for parallelization, which is ordered by achievable program speedup.

Cilkview [78] is a Cilk++ analyzer of program scalability in terms of number of cores. Cilkview performs system-level modeling of scheduling overheads and predicts program speedup. Bounds on the speedup are presented to programmers for further analysis.

Autopin [87] is an auto-configuration framework for finding the best mapping between system cores and threads. Using profile runs, Autopin exhaustively probes all possible mappings and finds the best pinning configuration in terms of performance.

StreamIt [75] is a language for creating streaming applications and can take advantage of parallelism present in data flow graph representation of applications, including task, pipeline, and data parallelism. However, StreamIt is mostly a synchronous streaming system, where static scheduling is performed based on compile-time analysis of filters written in the StreamIt language.

Alchemist [132] is a dependence profiling technique based on post-dominance analysis and is used to detect candidate regions for parallel execution. It is based on the observation that a procedure with few dependencies with its continuation benefits more from parallelization.

Task assignment in distributed computing has been an active research problem for decades. General task assignment is intractable. In [50], several programs with special structures are considered and the optimal assignment is found by using a graph theoretic approach.

There has been extensive research in the literature on compiler support for instruction-level or fine-grained pipelined parallelism [88]. In this work, we look at coarse-grained pipelining techniques that address the problem of decomposing an application into higher-level pieces that can execute in pipeline parallel.

Relevant to our study is the work in [63], which provides compiler support for coarse-grained pipelined parallelism. To automate pipelining, it selects a set of candidate filter boundaries (a middleware interface exposed by DataCutter [48]), determines the communication volume for these boundaries, and performs decomposition and code generation in order to minimize the execution time. To select the best filters, communication costs across each filter boundary are estimated by static program analysis and a dynamic programming algorithm is used to find the optimal decomposition.

A more detailed analysis of the differences of our work from others is given in Appendix 4.10.

4.9.1 Related Work on Profilers

There are generally two ways to implement a program profiler; statistical sampling and code instrumentation. While sampling is less disruptive to the base program, instrumentation-based profiling can obtain more accurate results.

OProfile [27] and DCPI [37] are representative of sampling-based profiling schemes. They use hardware performance counters to attain high frequency with fairly low overhead.

Code instrumentation for profiling can be applied statically (i.e., before program execution) or dynamically (i.e., during execution). In particular, static instrumentation code can be added to source code manually, automatically through compiler assist, or to the compiled binary via binary translation.

Gprof [76] is a hybrid profiler in the sense that both instrumentation and sampling are used. Static instrumentation keeps track of caller graph and the execution time is obtained by statistical sampling. Our profiler used in auto-pipelining is similar in the sense that

it uses both sampling and instrumentation, but the instrumentation is dynamically turned on/off within the SPL runtime, rather then being injected at compile-time.

4.10 Comparison to Related Systems

Compared to [63] our auto-pipelining scheme is executed during runtime, since stream processing applications can contain arbitrary user-defined operators and dynamic runtime behavior, making static analysis impractical. This leads to various different design choices: First, rather than relying on static program analysis, we use online profiling to estimate the system overheads and to discover system bottlenecks. Second, while offline decomposition can afford to use dynamic programming for global optimal filter/threaded port selection, our online approach relies on a greedy algorithm for exploring local optimality. Last, computing units in our pipelines do not need to be linearly chained.

Compared to StreamIt, SPL applications follow the asynchronous streaming model, where operator selectivity, cost, and behavior are not known at compile-time. The approach we presented in this chapter is thus completely dynamic, both in terms of profiling and optimization.

Compared to other existing work, our auto-pipelining solution is similar to Autopin in terms of run-time auto-configuration. However, our solution avoids exhaustive search by employing a novel optimization algorithm which is based on profiling statistics. In terms of automatically finding parallelism opportunities, our system is similar to Kremlin and Cilkview. However, the threaded ports exposed in the SPL runtime provide a flexible mechanism for optimization, only requiring operator-level profiling information. Thus we avoid the heavy profiling needs of systems like Kremlin and Cilkview. Finally, auto-pipelining requires no programmer intervention and is designed for data stream processing systems.

4.11 Conclusion

In this chapter, we have described an auto-pipelining solution for data stream processing applications. It automatically discovers pipeline and task parallelism opportunities in stream processing applications, and applies dynamic profiling and controlling to adjust the level of parallelism needed to achieve the best throughput. Our solution is transparent in the sense that no changes are required on the application source code. Our experimental evaluation shows that our solution is also effective, matching or exceeding the speedup that can be achieved via expert tuning. Our solution has been implemented on a commercial-grade data stream processing system.

CHAPTER V

CONCLUSION

5.1 Summary

In this dissertation, I explore the big-data system design for the cloud from the perspective of data security and high performance. I make the following technical contributions.

First, I proposed HINDEX, for secondary index support on top of write-optimized key-value stores. To update the index structure efficiently in the face of an intensive write stream, HINDEX synchronously executes append-only operations and defers the so-called index-repair operations which are expensive. The core contribution of HINDEX is a scheduling framework for deferred and lightweight execution of index repairs. HINDEX has been implemented and is currently being transferred to an IBM big data product.

Second, I proposed Auto-pipelining for automatic performance optimization of streaming applications on multi-core machines. The goal is to prevent the bottleneck scenario in which the streaming system is blocked by a single core while all other cores are idling, which wastes resources. To partition the streaming workload evenly to all the cores and to search for the best partitioning among many possibilities, I proposed a heuristic based search strategy that achieves locally optimal partitioning with lightweight search overhead. The key idea is to use a white-box approach to search for the theoretically best partitioning and then use a black-box approach to verify the effectiveness of such partitioning. The proposed technique, called Auto-pipelining, is implemented on IBM Stream S.

Third, I proposed ϵ-PPI, a suite of privacy preserving index algorithms that allow data sharing among unknown parties and yet maintaining a desired level of data privacy. To differentiate privacy concerns of different persons, I proposed a personalized privacy definition and substantiated this new privacy requirement by the injection of false positives

in the published ϵ-PPI data. To construct the ϵ-PPI securely and efficiently, I proposed to optimize the performance of multi-party computations which are otherwise expensive; the key idea is to use addition-homomorphic secret sharing mechanism which is inexpensive and to carry out the distributed computation in a scalable P2P overlay.

5.2 Future Work

5.2.1 Elastic Storage Services in the Cloud

One unique characteristic of the cloud is the pay-as-you-go business model which leads to highly dynamic workloads. To adapt to the dynamic workload needs in the cloud and to minimize overhead, elastic resource provisioning is required. While existing key-value stores are scalable in the sense of being easy to add/remove storage nodes, they are not innately elastic; Most of key-value stores do not support the self-adjust ability to adapt their system and resource provisioning to the changing workload. Thus, one of the future research is to extend key-value stores to elastically support the changing workloads.

Supporting elastic key-value stores is a challenging problem, since it requires monitoring dynamic cloud workloads and making correct prediction of the future workload. The monitoring component needs to be lightweight and not to interfere the original workload. Predicting the future trend may require complex learning mechanisms to deeply understand the current workload. In addition, given a correctly predicted workload, it is difficult to search for the best software configuration that optimizes the performance. Because the key-value stores are complex distributed software with a large number of tuning knobs, it presents a large search space for the optimal software configuration.

5.2.2 NewSQL for Real-Time Analytics

Supporting big data management and scalable SQL has recently attracted a great deal of attentions in industry and is pioneered by many projects in big companies including Google F1, IBM BigInsights and so on. While shared-nothing databases has been studied in the research community for long time, it is the sheer volume of big data, large scale of cloud

systems and unique requirement of modern Web 2.0 applications that make the problem of supporting SQL on big data (or NewSQL) distinctive. Given the recent popularity of NoSQL stores and sustained success of relational databases, it presents new system design opportunities and challenges to get the best from the both worlds; while the SQL based relational databases offers strong consistency and optimized query performance, the NoSQL stores excel in schema-free flexibility and scalability. It would be interesting to explore the NewSQL research by supporting SQL on top of NoSQL stores.

One open problem of the future NewSQL research is to extend existing NoSQL stores and to design and develop a middleware layer between the storage tier and application tier to support analytical and transactional workloads with interactive performance. In this regards, my HINDEX work is the first step towards NewSQL that supports specific kinds of SQL queries, that is, SQL with value-based query parameter in WHERE statement. Other interesting problems can be further explored such as intelligent mapping of applications to correct data infrastructures by analyzing the service level agreement and application program characteristics. A possible approach is to apply the scalable data management technique in P2P networks (e.g. P2P data indexing [121, 119, 122, 120]) to the context of key-value stores and clouds.

5.2.3 Secure Key-Value Stores

While key-value stores are usually deployed in the public cloud where trusts are limited, existing key-value store systems lack security support. When cloud customers are uploading their sensitive data to the cloud, they expect their data can be protected in terms of both data confidentiality and authenticity.

To provide data confidentiality, access control mechanism and data encryption are two common technology. It would be an interesting research area to support fine-grained access control on key-value stores, and to enforce the access rules on the encrypted data.

Data authenticity is equally important for protecting key-value data in the cloud. When

a cloud customer uploads her data to the cloud, she might want to guarantee that the data stays unchanged when she will retrieve it back in the future. Data authenticity guarantees such property. While research work has been initiated to study the authenticated key-value store [117, 103, 110, 67], there are various open research problems and opportunities that can be studied, including but not limited to high-performance data authentication in the presence of expensive encryption operations, verification of data freshness in the eventual consistent stores, and others.

REFERENCES

[1] "Biginsights: http://www-01.ibm.com/software/data/infosphere/biginsights/whats_new.html,"

[2] "Cilk++. http://software.intel.com/en-us/articles/intel-cilk-plus/." retrieved October, 2011.

[3] "Commonwell: http://www.commonwellalliance.org/."

[4] "Commonwell rls: http://www.commonwellalliance.org/services."

[5] "Coursera: https://www.coursera.org/,"

[6] "Gahin: http://www.gahin.org/."

[7] "Hipaa, http://www.cms.hhs.gov/hipaageninfo/,"

[8] "Hive: http://hive.apache.org/,"

[9] "http://cassandra.apache.org/,"

[10] "http://code.google.com/p/leveldb/,"

[11] "http://en.wikipedia.org/wiki/target_corporation#2013_security_breach,"

[12] "http://hadoop-hbase.blogspot.com/2012/10/musings-on-secondary-indexes.html,"

[13] "http://hadoop.apache.org/,"

[14] "http://hbase.apache.org/,"

[15] "https://blogs.apache.org/hbase/entry/coprocessor_introduction,"

[16] "https://issues.apache.org/jira/browse/cassandra-1016,"

[17] "https://issues.apache.org/jira/browse/cassandra-1311,"

[18] "http://www.emulab.net/,"

[19] "Huawei hbase index: https://github.com/huawei-hadoop/hindex,"

[20] "Hypertable index: http://hypertable.com/blog/secondary_indices_have_arrived,"

[21] "Lois. http://www.lois-space.net/." retrieved October, 2011.

[22] "Netty: http://netty.io."

[23] "Nextgate: http://www.nextgate.com/our-products/empi/,"

[24] "Nhin: http://www.hhs.gov/healthit/healthnetwork."

[25] "Openempi: https://openempi.kenai.com/,"

[26] "Openmp. http://www.openmp.org." retrieved October, 2011.

[27] "Oprofile. http://oprofile.sourceforge.net/about/." retrieved October, 2011.

[28] "Phoenix index: http://phoenix.incubator.apache.org/secondary_indexing.html,"

[29] "Prism, http://en.wikipedia.org/wiki/prism_(surveillance_program),"

[30] "Protobuf: http://code.google.com/p/protobuf/."

[31] "Studip, http://www.studip.de/,"

[32] "Twister: http://twister.net.co/,"

[33] ABADI, D., AHMAD, Y., BALAZINSKA, M., ÇETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., LINDNER, W., MASKEY, A., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., and ZDONIK, S., "The design of the Borealis stream processing engine," in *CIDR*, 2005.

[34] AGRAWAL, P., SILBERSTEIN, A., COOPER, B. F., SRIVASTAVA, U., and RAMAKRISHNAN, R., "Asynchronous view maintenance for vlsd databases," in *SIGMOD Conference*, pp. 179–192, 2009.

[35] ALIASGARI, M., BLANTON, M., ZHANG, Y., and STEELE, A., "Secure computation on floating point numbers," in *NDSS*, 2013.

[36] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., and VASUDEVAN, V., "Fawn: a fast array of wimpy nodes," in *SOSP*, pp. 1–14, 2009.

[37] ANDERSON, J.-A. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S.-T., SITES, R. L., VANDEVOORDE, M. T., WALDSPURGER, C. A., and WEIHL, W. E., "Continuous Profiling: Where have all the cycles gone?," in *SOSP*, pp. 1–14, 1997.

[38] ANDRADE, H., GEDIK, B., WU, K.-L., and YU, P. S., "Processing high data rate streams in System S," *JPDC*, vol. 71, no. 2, pp. 145–156, 2011.

[39] ARASU, A., BABU, S., and WIDOM, J., "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[40] ARASU, A., BABCOCK, B., BABU, S., DATAR, M., ITO, K., MOTWANI, R., NISHIZAWA, I., SRIVASTAVA, U., THOMAS, D., VARMA, R., and WIDOM, J., "STREAM: The Stanford stream data manager," *IEEE Data Engineering Bulletin*, vol. 26, no. 1, 2003.

[41] ARMBRUST, M., CURTIS, K., KRASKA, T., FOX, A., FRANKLIN, M. J., and PATTERSON, D. A., "Piql: Success-tolerant query processing in the cloud," *PVLDB*, vol. 5, no. 3, pp. 181–192, 2011.

[42] ARMBRUST, M., FOX, A., PATTERSON, D. A., LANHAM, N., TRUSHKOWSKY, B., TRUTNA, J., and OH, H., "Scads: Scale-independent storage for social computing applications," in *CIDR*, 2009.

[43] BAKER, J., BOND, C., CORBETT, J., FURMAN, J. J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., and YUSHPRAKH, V., "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, pp. 223–234, 2011.

[44] BAWA, M., BAYARDO, JR, R. J., AGRAWAL, R., and VAIDYA, J., "Privacy-preserving indexing of documents on the network," *The VLDB Journal*, vol. 18, no. 4, 2009.

[45] BAWA, M., JR., R. J. B., and AGRAWAL, R., "Privacy-preserving indexing of documents on the network," in *VLDB*, pp. 922–933, 2003.

[46] BAWA, M., JR., R. J. B., RAJAGOPALAN, S., and SHEKITA, E. J., "Make it fresh, make it quick: searching a network of personal webservers," in *WWW*, pp. 577–586, 2003.

[47] BEN-DAVID, A., NISAN, N., and PINKAS, B., "Fairplaymp: a system for secure multi-party computation," in *ACM Conference on Computer and Communications Security*, pp. 257–266, 2008.

[48] BEYNON, M. D., KURÇ, T. M., ÇATALYÜREK, Ü. V., CHANG, C., SUSSMAN, A., and SALTZ, J. H., "Distributed processing of very large datasets with DataCutter," *Parallel Computing Journal*, vol. 27, no. 11, pp. 1457–1478, 2001.

[49] BLACKWELL, T., HARRIS, J., and SELTZER, M. I., "Heuristic cleaning algorithms in log-structured file systems," in *USENIX Winter*, 1995.

[50] BOKHARI, S. H., "Assignment problems in parallel and distributed computing," in *Kluwer Academic Publishing*, 1987.

[51] CAO, N., WANG, C., LI, M., REN, K., and LOU, W., "Privacy-preserving multi-keyword ranked search over encrypted cloud data," in *INFOCOM*, pp. 829–837, IEEE, 2011.

[52] CHAMBERLAIN, B. L., CALLAHAN, D., and ZIMA, H. P., "Parallel programmability and the Chapel language," *IJHPCA*, vol. 21, pp. 291–312, 2007.

[53] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., and GRUBER, R., "Bigtable: A distributed storage system for structured data (awarded best paper!)," in *OSDI*, pp. 205–218, 2006.

[54] CHARLES, P., GROTHOFF, C., SARASWAT, V. A., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., and SARKAR, V., "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005.

[55] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., and YERNENI, R., "Pnuts: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[56] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., and SEARS, R., "Benchmarking cloud serving systems with ycsb," in *SoCC*, pp. 143–154, 2010.

[57] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., and OTHERS, "Spanner: Google's globally-distributed database,"

[58] CUZZOCREA, A. and BERTINO, E., "A secure multiparty computation privacy preserving olap framework over distributed xml data," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, (New York, NY, USA), pp. 1666–1673, ACM, 2010.

[59] D. GRIPPI, M. SALZBERG, R. S. and ZHITROMIRSKIY., I., "Diaspora*. http://www.joindiaspora.com,"

[60] DAMGÅRD, I., GEISLER, M., KRØIGAARD, M., and NIELSEN, J. B., "Asynchronous multiparty computation: Theory and implementation," in *Public Key Cryptography*, pp. 160–179, 2009.

[61] DEAN, J. and GHEMAWAT, S., "Mapreduce: Simplified data processing on large clusters," in *OSDI*, pp. 137–150, USENIX Association, 2004.

[62] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., and VOGELS, W., "Dynamo: amazon's highly available key-value store," in *SOSP*, pp. 205–220, 2007.

[63] DU, W., FERREIRA, R., and AGRAWAL, G., "Compiler support for exploiting coarse-grained pipelined parallelism," in *SC*, p. 8, 2003.

[64] DUVALL, S. L., FRASER, A. M., ROWE, K., THOMAS, A., and MINEAU, G. P., "Evaluation of record linkage between a large healthcare provider and the utah population database," *JAMIA*, vol. 19, no. e1, 2012.

[65] ESCRIVA, R., WONG, B., and SIRER, E. G., "Hyperdex: a distributed, searchable key-value store," in *SIGCOMM*, pp. 25–36, 2012.

[66] ET AL, M. M. A., "System R: A relational approach to data management," *ACM TODS*, vol. 1, no. 2, pp. 97–137, 1976.

[67] FELDMAN, A. J., ZELLER, W. P., FREEDMAN, M. J., and FELTEN, E. W., "Sporc: Group collaboration using untrusted cloud resources," in *OSDI*, pp. 337–350, 2010.

[68] GARCIA, S., JEON, D., LOUIE, C. M., and TAYLOR, M. B., "Kremlin: Rethinking and rebooting gprof for the multicore age," in *PLDI*, 2011.

[69] GEDIK, B. and ANDRADE, H., "A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM InfoSphere Streams," *Software: Practice and Experience*, 2012.

[70] GEDIK, B., ANDRADE, H., and WU, K.-L., "A code generation approach to optimizing high-performance distributed data stream processing," in *ACM CIKM*, 2009.

[71] GEDIK, B., ANDRADE, H., WU, K.-L., YU, P. S., and DOO, M., "Spade: The System S declarative stream processing engine," in *ACM SIGMOD*, 2008.

[72] GENTRY, C., "Fully homomorphic encryption using ideal lattices," in *STOC*, pp. 169–178, 2009.

[73] GEORGE, L., *HBase - The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011.

[74] GIACOMONI, J., MOSELEY, T., and VACHHARAJANI, M., "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *ACM PPoPP*, 2008.

[75] GORDON, M. I., THIES, W., and AMARASINGHE, S., "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS*, 2006.

[76] GRAHAM, S. L., KESSLER, P. B., and MCKUSICK, M. K., "gprof: A call graph execution profiler (with retrospective)," in *Best of PLDI*, pp. 49–57, 1982.

[77] HAWKING, D., "Overview of the trec-9 web track," in *TREC*, 2000.

[78] HE, Y., LEISERSON, C. E., and LEISERSON, W. M., "The cilkview scalability analyzer," in *ACM SPAA*, 2010.

[79] HENECKA, W., KÖGL, S., SADEGHI, A.-R., SCHNEIDER, T., and WEHRENBERG, I., "Tasty: tool for automating secure two-party computations," in *ACM Conference on Computer and Communications Security*, pp. 451–462, 2010.

[80] HUANG, Y., EVANS, D., KATZ, J., and MALKA, L., "Faster secure two-party computation using garbled circuits," in *USENIX Security Symposium*, 2011.

[81] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., and FETTERLY, D., "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys*, pp. 59–72, 2007.

[82] JAIN, N., AMINI, L., ANDRADE, H., KING, R., PARK, Y., SELO, P., and VENKATRAMANI, C., "Design, implementation, and evaluation of the linear road benchmark on the Stream Processing Core," in *ACM SIGMOD*, 2006.

[83] JEŘÁBEK, E., "Dual weak pigeonhole principle, Boolean complexity, and derandomization," *Annals of Pure and Applied Logic*, vol. 129, pp. 1–37, 2004.

[84] JR., G. L. S., "Parallel programming and code selection in fortress," in *ACM PPoPP*, 2006.

[85] KHANDEKAR, R., HILDRUM, K., PAREKH, S., RAJAN, D., WOLF, J. L., WU, K.-L., ANDRADE, H., and GEDIK, B., "COLA: Optimizing stream processing applications via graph partitioning," in *USENIX Middleware*, 2009.

[86] KISSNER, L. and SONG, D., "Privacy-preserving set operations," in *in Advances in Cryptology - CRYPTO 2005, LNCS*, pp. 241–257, Springer, 2005.

[87] KLUG, T., OTT, M., WEIDENDORFER, J., and TRINITIS, C., "Autopin: Automated optimization of thread-to-core pinning on multicore systems," *HiPEAC*, vol. 3, pp. 219–235, 2011.

[88] KRISHNAMURTHY, S. M., "A brief survey of papers on scheduling for pipelined processors," *ACM SIGPLAN Notices*, vol. 25, no. 7, pp. 97–106, 1990.

[89] KUZU, M., KANTARCIOGLU, M., DURHAM, E. A., TOTH, C., and MALIN, B., "A practical approach to achieve private medical record linkage in light of public resources," *JAMIA*, vol. 20, no. 2, pp. 285–292, 2013.

[90] KUZU, M., KANTARCIOGLU, M., INAN, A., BERTINO, E., DURHAM, E., and MALIN, B., "Efficient privacy-aware record integration," in *EDBT*, pp. 167–178, 2013.

[91] LAKSHMAN, A. and MALIK, P., "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[92] LI, M., YU, S., CAO, N., and LOU, W., "Authorized private keyword search over encrypted data in cloud computing," in *ICDCS*, pp. 383–392, 2011.

[93] LIANG, S. and VISWANATHAN, D., "Comprehensive profiling support in the Java virtual machine," in *COOTS*, pp. 229–242, 1999.

[94] LU, J. and CALLAN, J. P., "Content-based retrieval in hybrid peer-to-peer networks," in *CIKM*, pp. 199–206, 2003.

[95] MALKHI, D., NISAN, N., PINKAS, B., and SELLA, Y., "Fairplay - secure two-party computation system," in *USENIX Security Symposium*, pp. 287–302, 2004.

[96] MITZENMACHER, M. and UPFAL, E., *Probability and computing - randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[97] NARAYAN, A. and HAEBERLEN, A., "DJoin: differentially private join queries over distributed databases," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Oct. 2012.

[98] NEWMAN, T. B. and BROWN, A. N., "Use of commercial record linkage software and vital statistics to identify patient deaths," *Journal of the American Medical Informatics Association*, vol. 4, no. 3, pp. 233–237, 1997.

[99] O'NEIL, P. E., CHENG, E., GAWLICK, D., and O'NEIL, E. J., "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.

[100] OUSTERHOUT, J. K., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G. M., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., and STUTSMAN, R., "The case for ramclouds: scalable high-performance storage entirely in dram," *Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2009.

[101] PENG, D. and DABEK, F., "Large-scale incremental processing using distributed transactions and notifications," in *OSDI*, pp. 251–264, 2010.

[102] POPA, R. A., REDFIELD, C. M. S., ZELDOVICH, N., and BALAKRISHNAN, H., "Cryptdb: protecting confidentiality with encrypted query processing," in *SOSP*, pp. 85–100, 2011.

[103] POPA, R. A., LORCH, J. R., MOLNAR, D., WANG, H. J., and ZHUANG, L., "Enabling security in cloud storage slas with cloudproof," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 31–31, USENIX Association, 2011.

[104] REINDERS, J., *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[105] ROSENBLUM, M. and OUSTERHOUT, J. K., "The design and implementation of a log-structured file system," in *SOSP*, pp. 1–15, 1991.

[106] SEARS, R. and RAMAKRISHNAN, R., "blsm: a general purpose log structured merge tree," in *SIGMOD Conference*, pp. 217–228, 2012.

[107] SELTZER, M. I., BOSTIC, K., MCKUSICK, M. K., and STAELIN, C., "An implementation of a log-structured file system for unix," in *USENIX Winter*, pp. 307–326, 1993.

[108] SHUTE, J., VINGRALEK, R., SAMWEL, B., HANDY, B., WHIPKEY, C., ROLLINS, E., LITTLEFIELD, M. O. K., MENESTRINA, D., CIESLEWICZ, S. E. J., RAE, I., and OTHERS, "F1: A distributed sql database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, 2013.

[109] SONG, D. X., WAGNER, D., and PERRIG, A., "Practical techniques for searches on encrypted data," in *IEEE Symposium on Security and Privacy*, pp. 44–55, 2000.

[110] STEFANOV, E., VAN DIJK, M., JUELS, A., and OPREA, A., "Iris: a scalable cloud file system with efficient integrity checks," in *ACSAC*, pp. 229–238, 2012.

[111] "StreamBase Systems." retrieved October, 2011.

[112] SWEENEY, L., "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.

[113] TAN, W., TATA, S., TANG, Y., and FONG, L. L., "Diff-index: Differentiated index in distributed log-structured data stores," in *EDBT*, pp. 700–711, 2014.

[114] TANG, Y. and GEDIK, B., "Autopipelining for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 12, pp. 2344–2354, 2013.

[115] TANG, Y., LIU, L., and IYENGAR, A., "e-ppi: Searching information networks with quantitative privacy guarantee," *Gerogia Tech Technical Report GIT-CERCS-14-02*.

[116] TANG, Y., LIU, L., and IYENGAR, A., "e-ppi: Locator service in information networks with personalized privacy preservation," in *ICDCS*, 2014.

[117] TANG, Y., WANG, T., HU, X., SAILER, R., PIETZUCH, P., and LIU, L., "Outsourcing multi-version key-value stores with verifiable data freshness," in *ICDE(System demo)*, 2014.

[118] TANG, Y., WANG, T., and LIU, L., "Privacy preserving indexing for ehealth information networks," in *CIKM*, pp. 905–914, 2011.

[119] TANG, Y., XU, J., ZHOU, S., and LEE, W.-C., "m-light: Indexing multi-dimensional data over dhts," in *ICDCS*, pp. 191–198, 2009.

[120] TANG, Y., XU, J., ZHOU, S., LEE, W.-C., DENG, D., and WANG, Y., "A lightweight multidimensional index for complex queries over dhts," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 12, pp. 2046–2054, 2011.

[121] TANG, Y. and ZHOU, S., "Lht: A low-maintenance indexing scheme over dhts," in *ICDCS*, pp. 141–151, 2008.

[122] TANG, Y., ZHOU, S., and XU, J., "Light: A query-efficient yet low-maintenance indexing scheme over dhts," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 59–75, 2010.

[123] "TCMalloc: Thread-caching malloc." retrieved August, 2012.

[124] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTHONY, S., LIU, H., and MURTHY, R., "Hive - a petabyte scale data warehouse using hadoop," in *ICDE*, pp. 996–1005, 2010.

[125] WANG, C., CAO, N., LI, J., REN, K., and LOU, W., "Secure ranked keyword search over encrypted cloud data," in *ICDCS*, pp. 253–262, 2010.

[126] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., and JOGLEKAR, A., "An integrated experimental environment for distributed systems and networks," in *OSDI*, 2002.

[127] WRIGHT, M., ADLER, M., LEVINE, B. N., and SHIELDS, C., "An analysis of the degradation of anonymous protocols," in *NDSS*, 2002.

[128] YABANDEH, M. and FERRO, D. G., "A critique of snapshot isolation," in *EuroSys*, pp. 155–168, 2012.

[129] "S4 distributed stream computing platform." retrieved October, 2011.

[130] YAO, A. C.-C., "Protocols for secure computations (extended abstract)," in *FOCS*, pp. 160–164, 1982.

[131] ZERR, S., DEMIDOVA, E., OLMEDILLA, D., NEJDL, W., WINSLETT, M., and MITRA, S., "Zerber: r-confidential indexing for distributed documents," in *EDBT*, pp. 287–298, 2008.

[132] ZHANG, X., NAVABI, A., and JAGANNATHAN, S., "Alchemist: A transparent dependence distance profiling infrastructure," in *CGO*, pp. 47–58, 2009.

VITA

Yuzhe Tang was born in Changsha, Hunan Province, China. He received his Bachelor and Master degrees in computer science and engineering from Fudan University, Shanghai, China, in 2006 and 2009 respectively. He started his Ph.D. study in the College of Computing at Georgia Institute of Technology since August 2009. His research interests are broadly in distributed systems and information security. Specifically, his Ph.D. work has focused on building scalable secure systems for big data management in the cloud; he has lead research projects on a variety of topics including big data indexing, secure key-value stores, efficient privacy preserving computations, performance optimizations of stream applications and indexing in peer-to-peer networks. Most often, his research begins with a real-world problem, and ends with a practical solution that is implemented. He has done internships at IBM Research, NEC Labs and Microsoft Research.